



```
complement(int D, int V)

void main(void)
{
    int A[max][max], B[max][max];
    system("clear");
    printf("\n\tRandom Graph generated\n");
    printf("\n\tEnter number of vertices: ");
    scanf("%d", &vertex);
    generation(A, &vertex);
    printf("\n\tGenerated Random Graph\n");
    display(A, &vertex);
    printf("\n\tComplement Graph\n");
    complement(A, B, &vertex);
    display(B, &vertex);
}
```

A Practical Approach to Data Structure and Algorithm with Programming in C

Akhilesh Kumar Srivastava

VISIT...

LANZAROTE
Caliente.COM

A PRACTICAL APPROACH TO DATA STRUCTURE AND ALGORITHM WITH PROGRAMMING IN C

A PRACTICAL APPROACH TO DATA STRUCTURE AND ALGORITHM WITH PROGRAMMING IN C

Akhilesh Kumar Srivastava



www.arclerpress.com

A Practical Approach to Data Structure and Algorithm with Programming in C

Akhilesh Kumar Srivastava

Arcler Press

2010 Winston Park Drive,

2nd Floor

Oakville, ON L6H 5R7

Canada

www.arclerpress.com

Tel: 001-289-291-7705

001-905-616-2116

Fax: 001-289-291-7601

Email: orders@arclereducation.com

e-book Edition 2020

ISBN: 978-1-77407-429-9 (e-book)

This book contains information obtained from highly regarded resources. Reprinted material sources are indicated and copyright remains with the original owners. Copyright for images and other graphics remains with the original owners as indicated. A Wide variety of references are listed. Reasonable efforts have been made to publish reliable data. Authors or Editors or Publishers are not responsible for the accuracy of the information in the published chapters or consequences of their use. The publisher assumes no responsibility for any damage or grievance to the persons or property arising out of the use of any materials, instructions, methods or thoughts in the book. The authors or editors and the publisher have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission has not been obtained. If any copyright holder has not been acknowledged, please write to us so we may rectify.

Notice: Registered trademark of products or corporate names are used only for explanation and identification without intent of infringement.

© 2020 Arcler Press

ISBN: 978-1-77407-369-8 (Hardcover)

Arcler Press publishes wide variety of books and eBooks. For more information about Arcler Press and its products, visit our website at www.arclerpress.com

ABOUT THE AUTHOR



Akhilesh Kumar Srivastava is a Computer Science Graduate from K.N.I.T. Sultanpur and Post Graduate from Dr. APJ Abdul Kalam Technical University Lucknow. The author is GATE and UGC Net Qualified. He has written three books and several research papers in International/National Journals and presented papers in International Conferences. The author runs his own YouTube Educational Channel with substantial viewership across the globe. The author has also received several awards from academic institutes and industries like Infosys Ltd.

TABLE OF CONTENTS

	<i>List of Abbreviations.....</i>	<i>xiii</i>
	<i>Acknowledgment.....</i>	<i>xv</i>
	<i>Preface.....</i>	<i>xvii</i>
Chapter 1	Introduction to Data Structures	1
	1.1. Types Of Data Structures.....	2
	1.2. Categorization Of Data Structures.....	8
	Exercises.....	9
Chapter 2	Algorithm Writing.....	11
	2.1. Algorithm	12
	2.2. Pseudo Code Writing: Industry Pattern	12
	Exercises.....	19
Chapter 3	Complexity Of Algorithm.....	21
	3.1. What Is Complexity	22
	3.2. Space Complexity Of Recursive Algorithm.....	22
	3.3. Notations For Representing Complexity	24
	Exercises.....	29
Chapter 4	Array.....	31
	4.1. Definition And Types Of Array	32
	4.2. Primitive Operations On Array	32
	4.3. Application Problems Related To Array	34
	4.4. Index Formula.....	37
	Exercises.....	43
Chapter 5	Searching Algorithms	45
	5.1. What Is Searching?.....	46

5.2. Sequential Search (Linear Search)	46
5.3. Binary Search	48
5.4. Comparison Of Linear And Binary Search.....	51
5.5. Indexed Sequential Search.....	52
Exercises.....	56
Chapter 6 Sorting Algorithms	57
6.1. Bubble Sort.....	58
6.2. Insertion Sort	60
6.3. Selection Sort.....	64
6.4. Quick Sort	67
6.5. Merge Sort.....	74
6.6. Set Operations Using Array	80
6.7. Counting Sort.....	91
6.8. Radix Sort.....	94
Exercises.....	95
Chapter 7 Heap & Heap Sort.....	97
7.1. Heap	98
7.2. Insertion In Heap	102
7.3. Deletion In Heap	103
7.4. Heap As Priority Queue	104
7.5. Heap Sort	105
Exercises.....	109
Chapter 8 Matrix Operations.....	113
8.1. Matrix Transposition.....	114
8.2. Printing Matrix In Spiral Form	117
8.3. Matrix Addition	118
8.4. Matrix Multiplication	120
Exercise	122
Chapter 9 Strings	123
9.1. Application Problems Related To Strings	124
Exercises.....	137

Chapter 10	Stack	139
	10.1. Stack Basics	140
	10.2. Primitive Operations On Stack.....	141
	10.3. Applications Of Stack	145
	Exercises.....	183
Chapter 11	Recursion	187
	11.1. Concept Of Recursion	188
	11.2. Application Problems Related To Recursion	188
	11.3. Towers Of Hanoi.....	195
	Exercises.....	199
Chapter 12	Queue	205
	12.1. Queue Basics.....	206
	12.2. Linear Queue.....	206
	12.3. Circular Queue.....	211
	12.4. Priority Queues.....	217
	Exercises.....	222
Chapter 13	Linked List.....	225
	13.1. Linked List Basics.....	226
	13.2. Linear Linked List.....	227
	13.3. Related Functions	237
	13.4. Set Operations Using Linked List	258
	13.5. Polynomial Representation And Polynomial Addition	267
	13.6. Long Numbers Arithmetic	273
	13.7. Linked List Implementation Of Queue	275
	13.8. Linked List Implementation Of Stack.....	280
	13.9. Linked List Implementation Of Priority Queue	286
	13.10. Circular Linked List.....	290
	13.11. Doubly Linked List.....	302
	13.12. Circular Doubly Linked List	318
	Exercises.....	332
Chapter 14	Binary Tree.....	337
	14.1. Binary Tree Basics	338

14.2. Implementation Of Binary Tree	340
14.3. Primitive Operations In Binary Tree.....	341
14.4. Building Binary Tree From Given Traversals.....	347
14.5. Level Order Traversal In Binary Tree.....	348
14.6. Application Problems Related To Binary Tree	349
14.7. Expression Tree	356
14.8. Huffman Coding	358
Exercises.....	360
Chapter 15 Binary Search Tree	371
15.1. Binary Search Tree (Bst) Basics	372
15.2. Primitive Operations On Binary Search Tree (Bst).....	372
15.3. Height Balanced Search Tree: Avl Tree	379
Exercises.....	385
Chapter 16 B-Tree & B+-Tree	393
16.1. B-Tree.....	394
16.2. B+Tree.....	396
Exercises.....	399
Chapter 17 Graph.....	401
17.1. Graph Basics	402
17.2. Applications Of Graphs	405
17.3. Representing Graphs.....	406
17.4. Graph Traversal.....	407
17.5. Shortest Path	412
17.6. Transitive Closure.....	415
17.7. Spanning Tree	418
17.8. Topological Sort	421
Exercises.....	424
Chapter 18 Sparse Matrices	433
18.1. Sparse Matrix Basics	434
18.2. Types Of Sparse Matrix	434
18.3. Addition Of Two Sparse Matrices	438
Exercises.....	439

Chapter 19 Hashing.....	441
19.1. Direct Address Table	442
19.2. Hash Tables	442
19.3. Collision Resolution In Hash Table.....	444
Exercises.....	448
 Index.....	 451

LIST OF ABBREVIATIONS

BFS	breadth-first search
BST	binary search tree
DFS	depth-first search
FCFS	first come first serve basis
FIFO	first-in-first-out
LIFO	last-in-first-out
LSD	least significant digit

ACKNOWLEDGMENT

The journey of this book started from my classroom where my students Harshit Gupta, Praveen Gupta, Priyanka Singh, Purendra Srivastava, Sarthak Dhama, Rimjhim Gupta, Rishav Singh Mehta, Shrey Singh, Shrusty Singh, Vivek have helped to kick this project off and helped me till the completion. The inspiration for writing this book comes from my favorite teacher, mentor, and friend Professor Samir Srivastava (K.N.I.T. Sultanpur). I am indebted to him for his constant praise and guidance. I would also like to mention the role of professional and competitive environment provided by ABES Engineering College authorities, and Head of CSE Department, Dr. Shailesh Tiwari in making this project possible.

Acknowledgment would not be complete without mentioning the support of my loving wife Ankita to whom I dedicate this book too.

PREFACE

“An Algorithm must be seen to be believed.”

—Donald Knuth

The purpose of this book is to culminate a deep understanding of Data Structures and Algorithms among the students.

The book provides a thorough and comprehensive coverage of the fundamentals of data structures and the principles of algorithm analysis. The book introduces the concept of basic data structures, numerous data structure techniques used in the current paradigm, and the basics of algorithms. A structured approach is followed to explain the process of problem-solving. A theoretical description of the problem is followed by the underlying technique.

The book then proceeds to explain in detail each of the data structures and elaborates the methods associated with it. A detailed explanation is provided with each method, the algorithm, and the C code. These are then ably supported by an example followed by an algorithm, and finally the corresponding program in C language.

In conclusion, the aim of this book is to provide a deep understanding of all data structures and algorithms, and all the methods associated with the implementation and operation of all the Data Structures. The book tries to explain concepts in-depth, starting from very basic.

This book is aimed at serving engineering graduate students of computer science and postgraduate level courses of computer applications. This book also works as a catalyst to the students approaching for competitive examinations such as GATE, UGC, NET, PSUs, etc.

1

Introduction to Data Structures

CONTENTS

1.1. Types Of Data Structures.....	2
1.2. Categorization Of Data Structures.....	8
Exercises.....	9

In computer science, when we arrange data elements in a structured manner where each element is of the same data type and is related to all the elements directly or indirectly, their management, memory storing method & organization is called *data structures*. The data elements can be of any type, including integers, strings, structures, and classes.

Whenever a data structure is created, the primitive operations should also be written on those data structures. The primitive operations are the operations required to store delete and manipulate the contents of the data structure. Common primitive operations are:

- Insertion;
- Deletion;
- Traverse;
- Search;
- Sort etc.

1.1. TYPES OF DATA STRUCTURES

1.1.1. Arrays

An array is the collection of homogeneous data elements stored in contiguous memory locations.

A[8] = { 12, 8, 34, 19, 26, 31, -5, 78 }

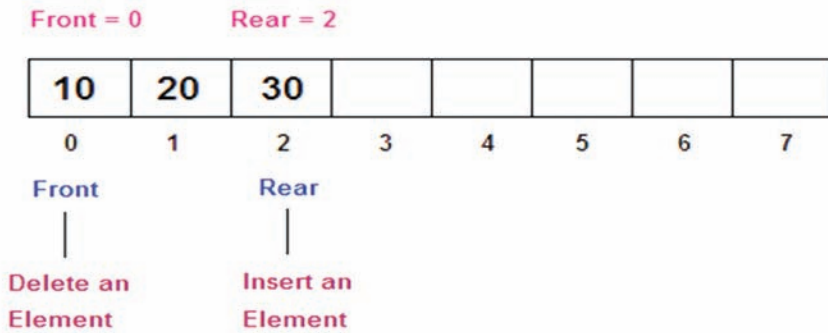
	1	2	3	4	5	6	7	8	Index
A	12	8	34	19	26	31	-5	78	
	1000	1002	1004	1006	1008	1010	1012	1014	Address

The array in C has indexes from 0–N–1 where N is the size of array. In the above diagram, indexes are starting from 1. This is a general array (not specific to any programming language).

They are the most basic data structures. They are used to store strings, implement databases, tables, vectors, and matrices.

1.1.2. Queues

Queues are the collection of data items of similar Types. They follow the principle of first-in-first-out (FIFO) or First Come First Serve (FCFS), which means that the first element added in the queue would be the first one to get removed. There are two ends in a Queue: REAR and FRONT. Insertions take place at REAR end and Deletions takes places from FRONT end.



They are used wherever we have a speed mismatch between the two entities out of which one is submitting the requests and other one handling those requests. e.g.,

- Input-output buffers;
- Printer buffers;
- CPU scheduling;
- Real-time booking systems, like flash sales and ticket booking systems;
- Round robin scheduling etc.;
- In real life, we can relate it to queue in front of the ticket window that offers tickets to the one who arrived first and in the same order as they arrived. The service here is in a First come first serve basis (FCFS);
- Automatic signaling at railway tracks;
- Packing of manufactured items at conveyer belt in a factory;
- It is used for the implementation of various algorithms like breadth-first search (BFS) in the Graph, Level order traversal in tree, etc.;
- Delivery of messages.

1.1.3. Priority Queues

Here, each element has its own priorities. For example, sorting integer elements, the priority of smaller elements will be more than the larger elements. They are also implemented using heaps.

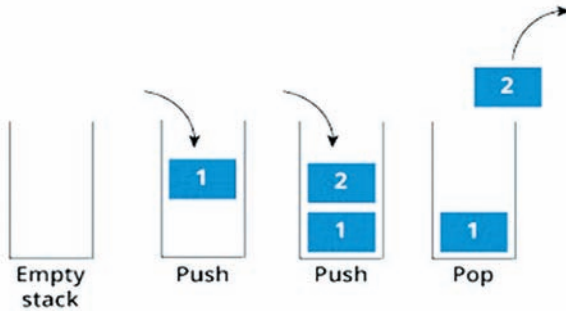
It is most widely used in:

- Creating sorted sequences like in bandwidth management;
- It is used for the implementation of Dijkstra Algorithm, Minimal Spanning Tree Prims Algorithm, Huffman Coding, etc.

1.1.4. Stacks

As the name suggests, they resemble real stacks of matter. They follow the principle

of last-in-first-out (LIFO) which means the first added element added would be the last one to be removed. Here, the access is only from the topmost element. There is only one end from which Insertions and Deletions can take place. It is called TOP.



Stacks are used by the system at various levels itself like:

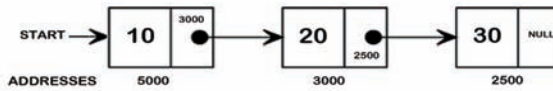
- Arithmetic expression evaluations and interconversions;
- Syntax parsing;
- Back tracing;
- Parentheses checking;
- Resolving function calls;
- Resolving recursion;
- Managing undo operations in the software;
- Visibility of received WhatsApp messages/emails etc.

1.1.5. Linked Lists

It is a linear collection of data elements. Here, their order is kept on record by dynamically storing the address of the next element instead of linear placements of elements and storing the address of the first element. The benefit of linked lists is that the size of the list can be changed until the whole memory has been utilized. Also, adding, and removing elements from between the nodes is less costly on computation.

They are the most basic data structure that can be used to implement almost every other data structures, including stacks, queues, graphs, trees, heaps, etc.

A linked list is the collection of nodes. Each node contains at least two fields. One field contains the data element, and the other one contains the address of the next node.



START keeps the address of the first node. Next field of the last node contains NULL

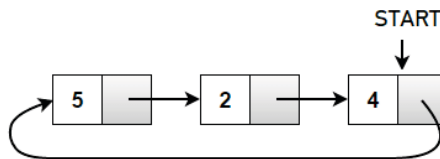
Linked lists are used by the system at various levels like:

- Dynamic partitioning during memory management;
- Keeping track of free and allocated disk blocks;
- Maintaining the process control block;
- Polynomial arithmetic;
- Long number arithmetic;
- Implementation of stack, queue, priority queue, tree, graph, tries, etc.

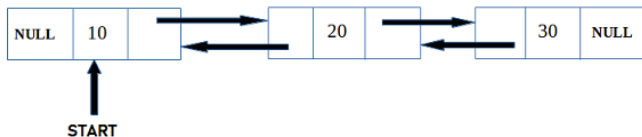
There are various types of linked lists:

Linear Linked List: Above diagram shows linear Linked List where each node contains the address of the next node, and the last node contains no address.

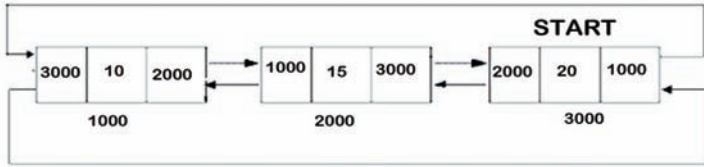
Circular Linked List: Circular Linked List is more like Linear Linked List except that the next field of the last node contains the address of the first node.



Doubly Linked List: Each node in this type of the linked list contains three fields. One contains information. One is used for the address of the previous node another one is used for the address of the next node. Binary Trees are implemented using the doubly linked list.

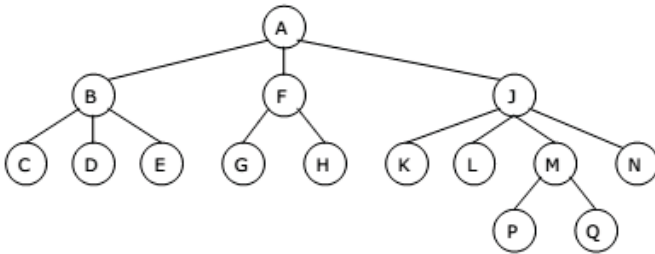


Circular Doubly Linked List: As in Doubly, Each node in this type of the linked list contains three fields. One contains information. One is used for the address of the previous node another one is used for the address of the next node. START keeps the address of the last node as in the circular linked list.



1.1.6. Trees

It is a linked data structure which mimics a hierarchical tree structure with a root node and having subtrees which can have multiple children. The terminal, i.e., last elements of trees are referred to as leaves. Between any two elements, there is a unique path. They are directional.

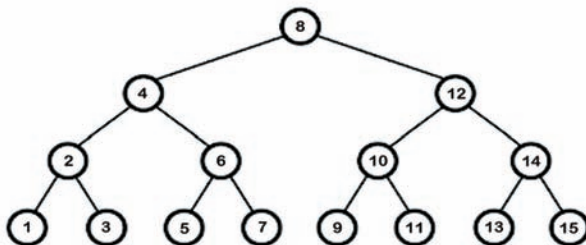


In computer science, the tree data structure can be used for:

- Storing the directory structure;
- Building the parse tree (During Syntax Analysis in a Compiler);
- Search space tree etc.

1.1.7. Binary Trees

These are a special kind of trees where each node can have a maximum of two children. As a definition, The tree elements are treated as a set. This set can be partitioned into three disjoint subsets. One of which is the root. Other two are themselves the binary trees called Left and Right subtrees.



In the above tree, $S=\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$

$S1=\{8\}$

$S2=\{1, 2, 3, 4, 5, 6, 7\}$

$S3=\{9, 10, 11, 12, 13, 14, 15\}$

Binary tree are used in various computer science applications e.g.,

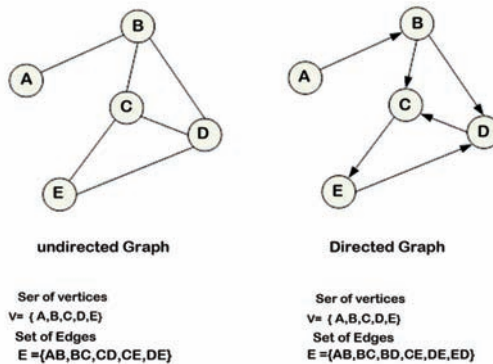
- Data compression through Huffman coding;
- Representation of arithmetic expression;
- Searching (using binary search tree (BST));
- Sorting (using heap);
- Making decision trees etc.

Special types of binary trees:

- Heaps;
- BST;
- AVL tree;
- Interval tree;
- Splay tree.

1.1.8. Graphs

The graph is used for the representation of the connection between some stations. It consists of a set of vertices and a set of edges.



The graph has a wide range of applications in general life and computer science for:

- Finding shortest path between a pair of station (source and destination) using GPS;
- Path search;

- Travelling salesman problem;
- Maximum flow problem;
- Finding minimal spanning tree;
- Chinese postman problem;
- Finding data flow path in computer networks;
- Order of activities using topological sort etc.

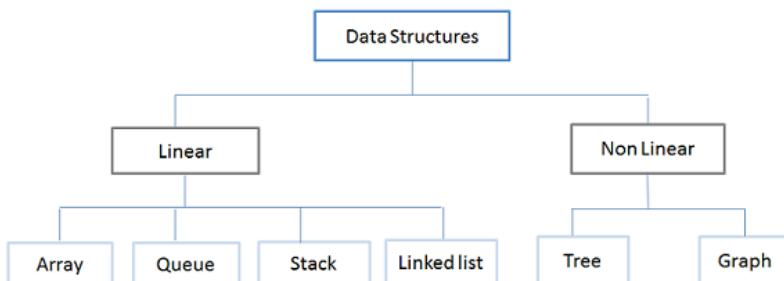
1.2. CATEGORIZATION OF DATA STRUCTURES

1.2.1. Linear Data Structure

The data structure is considered to be linear if the data elements construct a sequence of a linear list. The elements are adjacently attached to each other and in a specified order. The examples included in the linear data structure are array, stack, queue, linked list, etc. The traversal of data elements and Insertion or deletion are done sequentially.

1.2.2. Non-Linear Data Structure

In this, the data elements can be attached to more than one element exhibiting the hierarchical relationship, which involves the relationship between the child, parent, and grandparent. In the non-linear data structure, the traversal of data elements and Insertion or deletion are not done sequentially. There are two common examples of the non-linear data structure – tree and graph. A tree data structure organizes and stores the data elements in a hierarchical relationship.



EXERCISES

1. **What are the data structures? State the different types of data structures and examples of each.**
2. **State the differences between linear and non-linear data structures.**
3. **Define:**
 - a. Stack;
 - b. Queue;
 - c. Binary tree;
 - d. Graph.
4. **Explain the applications of various data structures**
5. **Pick the odd one out**
 - a. Composite data type
 - b. User defined data type
 - c. stack
 - d. Array
6. **Pick the odd one out**
 - a. Data type
 - b. Data object
 - c. Abstract data type
 - d. Data line

2

Algorithm Writing

CONTENTS

2.1. Algorithm	12
2.2. Pseudo Code Writing: Industry Pattern	12
Exercises	19

2.1. ALGORITHM

Representation of a solution to any problem is termed as an algorithm. Algorithms can be written in the form of:

- Flow chart (pictorial representation);
- In own words;
- Step by step instructions;
- Pseudocode.

Algorithms are written as the general solution to any problem. It is not written specifically to any Programming Language.

2.2. PSEUDO CODE WRITING: INDUSTRY PATTERN

For writing any algorithm in Industry pattern, the format is:

ALGORITHM<Name of Algorithm> (Argument List)

INPUT:<Description of Input in words>

OUTPUT:<Description of Output in words>

BEGIN:

Instructions

END;

Name of the algorithm is written in camel casing form. i.e., for multiple words in the name of the algorithm, the first name written in upper case and rest are written in lower case letters.

Example:

- SumOfDigitsOfNumber;
- FibonacciSeries;
- BubbleSort;
- EvenOrOdd.

2.2.1. Rules for Writing Various Syntax

Type	Pattern Used		Equivalent C Language Term
Input	READ	scanf()	
Output	WRITE	printf()	
Conditional ELSE	IF <condition> THEN	if(condition)	
	Else		

Loop FOR Lower limit TO Upper Limit STEP <n> DO DO --- WHILE <Con- dition> BREAK	WHILE <condition> DO	while (Condi- tion) { }	
	for(initialization; condition; increment/ decrement)		
	do { } while(Condition);		
	break;		

While writing the conditional and loop blocks, no braces are applied. Proper indentation is provided to make the algorithm readable. For assignment ← is used. Rest of the opearators are same as that in C.

- **Writing Conditional Statements:** For writing conditional Statements, IF is written followed by condition followed by THEN. In case of false condition ELSE can be written.

```
IF condition THEN
Statement1
Statement2
Statement3
...
ELSE
Statement4
```

- **Writing Loops**

```
For WHILE loop
WHILE Condition DO
Statement1
Statement2
Statement3
...
```

```
For FOR Loop
FOR Condition DO
Statement1
Statement2
Statement3
...
```

Here it is assumed that loop counter increments by one after each execution. In case of loop counter increment is not 1, below given pattern is followed:

```
For reverse loop and loop counter is decremented by 1 after each execution
FOR Condition STEP -1 DO
Statement1
```

Statement2

Statement3

...

For reverse loop and loop counter is decremented by 2 after each execution

FOR Condition STEP -2 DO

Statement1

Statement2

Statement3

...

For forward loop and loop counter is incremented by 2 after each execution

FOR Condition STEP +2 DO

Statement1

Statement2

Statement3

...

For forward loop and loop counter is multiplied with 2 after each execution

FOR Condition STEP *2 DO

Statement1

Statement2

Statement3

...

For reverse loop and loop counter is divided with 2 after each execution

FOR Condition STEP /2 DO

Statement1

Statement2

Statement3

...

2.2.2. Examples of Some Algorithms

ALGORITHM EvenOrOdd()

INPUT: An integer Number

OUTPUT: Nature of Number, i.e., Even or Odd

BEGIN:

READ(N)

IF $N \% 2 == 0$ THEN

WRITE("EVEN NUMBER");

ELSE

WRITE("ODD NUMBER");

END;

ALGORITHM LeapOrNonLeap(year)

INPUT: An year

OUTPUT: Nature of year, i.e., Leap or Non Leap

BEGIN:

IF $\text{year} \% 4 == 0$ AND $\text{year} \% 100 \neq 0$ OR $\text{year} \% 400 == 0$ THEN

WRITE("Leap year")

ELSE

WRITE("Non Leap Year")

END;

ALGORITHM TableOfNumber()

INPUT: An Integer Number

OUTPUT: Table of Input Number

BEGIN:

READ (N)

FOR $i \leftarrow 1$ TO 10 DO

WRITE ($N*i$)

END;

ALGORITHM NumberReverse()

INPUT: An Integer Number

OUTPUT: Reverse of Input Number

BEGIN:

READ (N)

Rev $\leftarrow 0$

WHILE $N > 0$ DO

Remainder $\leftarrow N \% 10$

Rev $\leftarrow \text{Rev} * 10 + \text{Remainder}$

$N \leftarrow N / 10$

WRITE ("Reversed Number is")

WRITE (Rev)

END;

ALGORITHM PrimeNumber()

INPUT: A range of Numbers

OUTPUT: All Prime Numbers

BEGIN:

READ (LowerLimit)

READ (UpperLimit)

FOR i $\leftarrow$ LowerLimit TO UpperLimit Do

FOR j $\leftarrow$ 2 TO SQRT(i) DO

IF i%j == 0 THEN

 BREAK;

IF J $\leftarrow$ SQRT(i)+1 THEN

WRITE(i)

END;

ALGORITHM BubbleSort (A[], N)

INPUT: A set of N Numbers

OUTPUT: Sorted Numbers

BEGIN:

FOR I $\leftarrow$ 1 TO n-1

FOR j $\leftarrow$ 0 TO n-i-1 DO

IF A[j]>A[j+1] THEN

CALL Exchange(A[j], A[j+1])

END;

ALGORITHM Exchange (a, b)

INPUT: Two Numbers

OUTPUT: Swapped Values of Numbers

BEGIN:

t $\leftarrow$ a

a $\leftarrow$ b

b $\leftarrow$ t

END;

ALGORITHM Factorial (N)

INPUT: An Integer Number

OUTPUT: Factorial Value of the Number

BEGIN:IF $N \leftarrow 0$ THEN

RETURN 1

ELSE

RETURN $N * \text{CALL Factorial}(N-1)$ **END;****ALGORITHM FibonacciNumber (N)**

INPUT: A integer Numbers

OUTPUT: Nth Fibonacci Term

BEGIN:IF $N == 1$ THEN

RETURN 0

ELSE

IF $N == 2$ THEN

RETURN 1

ELSE

RETURN $\text{CALL Fibonacci}(N-1) + \text{CALL Fibonacci}(N-2)$ **END;****ALGORITHM BinarySearch (A[], N, SearchKey)**

INPUT: An Array of Size N and a search key

OUTPUT: Location of Search key in the given Array

BEGIN:Low $\leftarrow 0$ High $\leftarrow N-1$ WHILE Low $\leq$ HighMid $\leftarrow (Low+High)/2$ IF $A[\text{Mid}] == \text{SearchKey}$ THEN

RETURN Mid

```
ELSE
IF SearchKey<A[Mid] THEN
High  $\leftarrow$  Mid-1
ELSE
Low  $\leftarrow$  Mid+1
```

```
RETURN -1
```

```
END;
```

For Calling a Function, we usually apply a keyword CALL

Recursive Binary Search

ALGORITHM BinarySearch (A[], Low, High SearchKey)

INPUT: An Array of Size N and a search key

OUTPUT: Location of Search key in the given Array

BEGIN:

```
IF Low <= High
```

```
Mid  $\leftarrow$  (Low+High)/2
```

```
IF A[Mid] == SearchKey THEN
```

```
RETURN Mid
```

```
ELSE
```

```
IF SearchKey<A[Mid] THEN
```

```
CALL BinarySearch(A[ ], low, Mid-1, SearchKey)
```

```
ELSE
```

```
CALL BinarySearch(A[ ], Mid+1, High, SearchKey)
```

```
RETURN -1
```

```
END;
```

EXERCISES

1. **What is an algorithm? Differentiate between an algorithm and a pseudo code.**
2. **Write an algorithm for the following problems:**
 - a. To find the factorial of a number.
 - b. To find the sum of digits.
 - c. To find a number in an array by linear search.
3. **Differentiate between a for-loop, while-loop, and a do-while loop.**

Programming Questions

1. **Write a Program (here and after abbreviated as WAP) to find the sum of digits.**
2. **WAP to exchange the value of two integer variables without using a third variable. (Hint: You can use addition or XOR operator)**

3

Complexity of Algorithm

CONTENTS

3.1. What Is Complexity	22
3.2. Space Complexity Of Recursive Algorithm.....	22
3.3. Notations For Representing Complexity	24
Exercises.....	29

3.1. WHAT IS COMPLEXITY

Efficiency measure of any algorithm is termed as the complexity.

The algorithm, when implemented in the form of program, is expected to consume some computer resources. CPU and RAM are the two most important computer resources.

The complexity of any algorithm is measured in two terms:

1. **Time Complexity:** It is the count of the number of instructions of an Algorithm required to be executed to implement the logic.
2. **Space Complexity:** It is the amount of extra space required for the Execution of the Algorithm.

For iterative Algorithms, the number of local variables in an Algorithm, Array etc taken are counted in the space complexity.

e.g., Consider the simple algorithm for swapping two numbers.

ALGORITHM Exchange (a, b)

INPUT: Two Numbers

OUTPUT: Swapped Values of Numbers

BEGIN:

t $\leftarrow$ **a**

a $\leftarrow$ **b**

b $\leftarrow$ **t**

END;

where a and b are the variables which are given as parameter, t is the only extra variable taken for implementation of the logic. Hence the space complexity is counted as 1 for this algorithm. In case an array of size 10 and a loop counter is taken in the algorithm, space complexity is counted as 11.

3.2. SPACE COMPLEXITY OF RECURSIVE ALGORITHM

In the recursive algorithm, a maximum number of activation records pending at any moment is counted as the space complexity.

1. **Activation Records:** When a function is called, an activation record is maintained in the STACK portion in the process boundary. The Activation records contain many information. These are mentioned in the table given below:

Return Value

Local Variables
Formal Arguments
Actual Arguments
Return Address

2. **Return Address:** Denotes the address at which the control will return upon termination of the function.
3. **Actual Arguments:** The data values passed as arguments to the function
4. **Formal Arguments:** The variables in which the actual arguments are received in the function definition.
5. **Local Variables:** The additional variables taken in function to implement the logic.
5. **Return Value:** Output of a function which will be returned to the calling function.

Take an example of a simple C-program segment:

```
void main()
{
int x=10, y=20, z;
z=Sum(x, y);
printf("Sum of data values is %d, "z);
}
```

```
Sum(int a, int b)
```

```
{
int s;
s=a+b;
return s;
}
```

For the above C Program, Assume the address of statement at 4 in RAM is 10000. Then the content of activation record will be:

Return Value	30
Local Variable	int s

Formal Arguments	int a, int b
Actual Arguments	x, y
Return Address	10000

In case the local variables are constant in function definition, the size of activation record is assumed to be constant. While computing the space complexity, size of 1 activation record is counted as 1 space.

Consider the below-mentioned algorithm for finding the factorial of an algorithm:

ALGORITHM Factorial (N)

INPUT: An Integer Number

OUTPUT: Factorial Value of the Number

BEGIN:

IF $N \leftarrow 0$ THEN

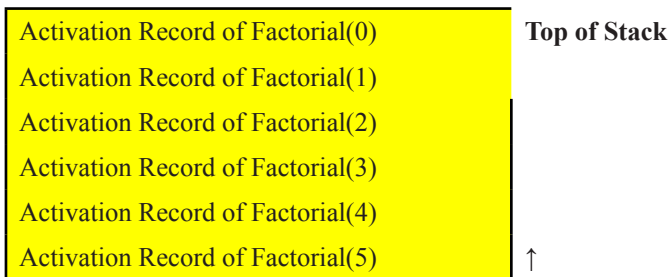
RETURN 1

ELSE

RETURN $N * \text{CALL Factorial}(N-1)$

END;

In case the call is for Factorial (5), the Stack will have following activation records pending to the maximum at any moment.



Here 6 activation records are pending and it will start decreasing once recursive calls with base condition is made ($N=0$ in this case).

The space complexity of this algorithm is $N+1$ as $N+1$ activation records are pending at any moment to the maximum.

3.3. NOTATIONS FOR REPRESENTING COMPLEXITY

There are three notations in which Time and space complexity can be represented.

Notation	Pronunciation	Represents
O	Big Oh	For representing Worst Case
Ω	Big Omega	For representing Best Case
Θ	Big Theta	For representing Average case or when there is no separated Best and Worst Case

In case there are fixed number of statements are to be executed for any input, The Time Complexity is considered to be Constant and Written in Θ (Theta) Notation. $\Theta(1)$ means the constant Time complexity.

In case there are fixed number of variables for Algorithm, space complexity is assumed to be constant and written in Θ (Theta) Notation. $\Theta(1)$ means the constant space complexity.

Number of Statements for Any Input	Time Complexity
4	$\Theta(1)$
5	$\Theta(1)$
7	$\Theta(1)$
1	$\Theta(1)$

ALGORITHM EvenOrOdd()

INPUT: An integer Number

OUTPUT: Nature of Number, i.e., Even or Odd

BEGIN:

READ(N)

IF $N \% 2 == 0$ THEN

WRITE("EVEN NUMBER");

ELSE

WRITE("ODD NUMBER");

END;

Total number of statements for any value of N is 2. Hence Time Complexity is $\Theta(1)$.

Total number of variables used in the logic here is 1. Hence Space Complexity is $\Theta(1)$.

ALGORITHM TableOfNumber()

INPUT: An Integer Number

OUTPUT: Table of Input Number

BEGIN:

READ (N)

FOR $i \leftarrow 1$ TO 10 DO

WRITE ($N*i$)

END;

Total number of statements for any value of N is 10 (as loop iterates fixed 10 times and one statement is inside the loop). Hence Time Complexity is $\Theta(1)$.

Total number of variables used in the logic here is 2. Hence Space Complexity is $\Theta(1)$.

In case there are variable number of statements, the complexity is written with respect to variable number. Addition, subtraction, and Multiplication of a Number is omitted while writing the complexity. Complexity is written w.r.t. dominating factor only.

Number of Statements for Any Input	Time Complexity
N	$\Theta(N)$
$N+2$	$\Theta(N)$
$N-1$	$\Theta(N)$
$N+7$	$\Theta(N)$
$2*N$	$\Theta(N)$
$2*N+4$	$\Theta(N)$
$2/3*N-7$	$\Theta(N)$

ALGORITHM Factorial (N)

INPUT: An Integer Number

OUTPUT: Factorial Value of the Number

BEGIN:

$F \leftarrow 1$

IF $F == 0$ THEN

RETURN 1

ELSE

FOR $i \leftarrow 1$ TO N Do

$F \leftarrow F*i$

RETURN F

END;

The above algorithm requires N multiplications, 1 initialization to F and one RETURN Statement. Hence total statements which are to be executed are $N+2$. Complexity of this algorithm is $\Theta(N)$.

Total number of variables used in the logic here is 2. Hence Space Complexity is $\Theta(1)$.

Consider the Algorithm given below for sorting the N numbers:

ALGORITHM Function (A[], N)

INPUT: A set of N Numbers

OUTPUT: Adjustment of array elements

BEGIN:

FOR $i \leftarrow 1$ TO $n-1$

FOR $j \leftarrow 0$ TO $n-i-1$ DO

IF $A[j] > A[j+1]$ THEN

$T \leftarrow A[j]$

$A[j] \leftarrow A[j+1]$

$A[j+1] \leftarrow T$

END;

Total number of statements which are to be executed are:

For each execution of outer loop, there will be $n-i$ executions in inner loop. Inner loop contains 3 statements.

Total statements to be executed are:

$$= (n-1) + (n-2) + (n-3) + \dots + 1$$

$$= n(n-1)/2$$

$$= n^2/2 - n/2$$

Here it is assumed that n is very large quantity hence n^2 will be even larger. Complexity is written w.r.t. dominating factor, i.e., n^2 . So complexity of above algorithm is $\Theta(n^2)$

Total number of variables used in the logic here is 3. Hence Space Complexity is $\Theta(1)$.

3.3.1. Best and Worst Case Complexity

Consider an algorithm given below which searches an element in an array.

ALGORITHM Linear_search(A[], N, key)

BEGIN:

FOR i ← 1 TO N DO

IF A[i] == key THEN

RETURN i

RETURN -1

END;

If the searching key is found at first position in the array, in the comparison inside the loop the index of array is returned. This is the best case. Total number of loop execution here in this case will be 1 hence Best Case Time Complexity is written in omega notation: (1)

In worst case, the element will be found at last location hence loop will iterate for N time. This is the worst case. Worst case complexity is written in Big Oh notation: O(N)

3.3.2. Average Case Complexity

For finding the average case complexity, it is equally likely that search element can be found at any location. So the probability of successful search at each location is $1/N$. The Complexity can be found as

Probability * number of search required for successful search

Complexity = $1/N * 1 + 1/N * 2 + 1/N * 3 + \dots + 1/N * N$

= $1/N * (1+2+3+\dots+N)$

= $1/N * (N*(N+1)/2)$

= $(N+1)/2$

= $N/2 + 1/2$

Complexity can be written as $\Theta(N)$

EXERCISES

- 1 Define big-O notation of complexity. Why is it called as an asymptotic notation?
- 2 Arrange the following complexities in ascending order of asymptotic growth:

c^n , $n^{\cos(n)}$, 2^n , $128^{\log(n)}$, $\log(n^n)$

where, $c > 1$

3. State the importance of time and space complexities in real world.
4. Arrange the following functions, often used to represent complexity of algorithms in order from slowest to fastest. $O(1)$, $O(n)$, $O(n \cdot \log_2 n)$, $O(\log_2 n)$, $O(n^2)$, $O(2^n)$
5. Find the time complexities of the following code fragments:

a. `for(x=0;x<n;x++)`

`for(y=0;y<m;y++)`

`//constant time operation`

b. `for(x=0;x<n;x++)`

`for(y=1;y<n;y*=2)`

`//constant time operation`

c. `for(x=0;x<n;x++)`

`for(y=0;y<n;y++)`

`x=x+1;`

d. `void test(int n)`

```
{
return n*test(n-1);
}
```

e. `void test(int n)`

```
{
return test(n-1)+test(n-1)+1;
}
```

f. `for(x=0;x<n;x++)`

`for(y=0;y<n-x;y++)`

`//constant time operation`

g. `for(x=0;x<n;x*=2)`

`//constant time operation`

h. `for(x=n; x>0; x/=2)`

`//constant time operation`

```
        i.    for(x=0;x<n;x++)
for(y=n;y>0; y/=2)
//constant time operation

        j.    for(x=0;x<n;x*=2)
for(y=0;y<n;y++)
//constant time operation
```

Programming Questions

1. **Given an array with only 0's and 1's, WAP to shift all the 1's to the right side and 0's to the left side.**
2. **WAP to find the nth number of the Fibonacci sequence both iteratively and recursively.**
3. **WAP to find out if a number is in power of two. Try out different approaches, e.g., pre-calculation, bit manipulation, etc.**

For all the above programs, state time and space complexity of your program. Can the complexities be reduced? Tabulate the complexities of different approaches.

4

Array

CONTENTS

4.1. Definition And Types Of Array	32
4.2. Primitive Operations On Array	32
4.3. Application Problems Related To Array	34
4.4. Index Formula.....	37
Exercises	43

4.1. DEFINITION AND TYPES OF ARRAY

Definition: Array is the collection of homogeneous data elements stored in contiguous memory locations.

$$A[8] = \{12, 8, 34, 19, 26, 31, -5, 78\}$$

	1	2	3	4	5	6	7	8	Index
A	12	8	34	19	26	31	-5	78	
	1000	1002	1004	1006	1008	1010	1012	1014	Address

In the above array, eight size array is declared. Index of array is starting from 1 in Algorithm notations (it starts from 0 in case of C or Java). Here it is assumed that each element requires 2 bytes for storage. The location of subsequent element is contiguous.

A one-dimensional array is like a list.

A two-dimensional array is like a table or a Matrix

$$A[3][5] = \{5, 12, 17, 9, 3, \\ 13, 48, 14, 1, \\ 9, 6, 3, 7, 21\}$$

		Columns →				
		1	2	3	4	5
Rows ↓	1	5	12	17	9	3
	2	13	4	8	14	1
	3	9	6	3	7	21

2D Array of size 3 × 5

4.2. PRIMITIVE OPERATIONS ON ARRAY

1. Traversal

ALGORITHM Traverse(A[], N)

BEGIN:

FOR $i \leftarrow 1$ TO N DO

WRITE(A[i])

END;

Time Complexity: $\Theta(N)$

The above algorithm requires N multiplications. Hence, the number of statements to be executed are N.

Space Complexity: $\Theta(1)$

The only extra variable taken here is i.

2. Insertion

ALGORITHM Insertion(A[], N, i, key)**BEGIN:**FOR j $\leftarrow$ N TO i STEP-1 DOA[j+1] $\leftarrow$ A[j]A[i] $\leftarrow$ keyN $\leftarrow$ N+1**END;****Time Complexity: $O(N)$, $\Omega(1)$**

When the element is to be inserted in the beginning, N number of shiftings will be required and two statements to assign the value and increase the value of N. Hence, N+2 statements will be executed (**Worst Case**)

When the element is to be inserted at the end, no shifting is required. Only two statements will be executed. (**Best Case**)

Space Complexity: $\theta(1)$

The only extra variable taken here is j

3. Deletion

ALGORITHM Deletion(A[], N, i)**BEGIN:**X $\leftarrow$ A[i]FOR j $\leftarrow$ i+1 TO N DOA[j-1] $\leftarrow$ A[j]N $\leftarrow$ N-1

RETURN x

END;**Time Complexity:** $O(N)$, $\Omega(1)$

When the element is to be Deleted from the beginning, N number of shiftings will be required and one statement is required to decrease the value of N. Hence, N+1 statements will be executed (Worst Case)

When the element is to be deleted from the end, no shifting is required. Only one statement will be executed. (Best Case)

Space Complexity: $\theta(1)$

The only variable taken here is X

#include <stdio.h>

void Traverse(int A[], int n){

```

for (int i=1; i <= n; i++) {
printf(“%d”, A[i-1]);
}
}

```

```

void Insertion(int A[ ], int N, int i, int key){
for (int j=N; j >= i; j--) {
    A[j+1]=A[j];
}
A[i]=key;
N=N+1;
}

```

```

int deletion(int A[ ], int N, int i){
int X=A[i];
for (int j=i+1; j<N; j++) {
    A[j-1]=A[j];
    N=N-1;
}
return X;
}
int main(int argc, constchar * argv[ ]) {
int A[ ]={1, 2, 3, 4, 5, 6, 7};
Traverse(A, 4);
Insertion(A, 7, 3, 7);
deletion(A, 7, 3);
}

```

4.3. APPLICATION PROBLEMS RELATED TO ARRAY

1. Insertion in sorted array

ALGORITHM InsertionSortedArray(A[], N, key)

BEGIN:

$i \leftarrow 0$

```
WHILE A[i]<key DO
```

```
  i ← i+1
```

```
FOR j ← N-1 TO i STEP-1 DO
```

```
  A[j+1] ← A[j]
```

```
  A[i] ← key
```

```
  N ← N+1
```

```
END;
```

Time Complexity: $\Theta(N)$

The while loop runs i number of statements, the for loop runs $n-i$ statements and three statements for assigning the value of i , $A[i]$, and incrementing the value of N . In any case, $i+n-i+3=n+3$ statements are executed.

Space Complexity: $\Theta(1)$

Here, only two variables i and j are used.

```
#include <stdio.h>
```

```
void Traverse(int A[ ], int n){
```

```
  for (int i=1; i <= (n+1); i++) {
```

```
    printf("%d", A[i-1]);
```

```
  }
```

```
}
```

```
void InsertionSortedArray(int A[ ], int N, int key){
```

```
  int i=0;
```

```
  while (A[i]<key) {
```

```
    i=i+1;
```

```
  }
```

```
  for (int j=N-1; j >= i; j--) {
```

```
    A[j+1]=A[j];
```

```
  }
```

```
  A[i]=key;
```

```
  N=N+1;
```

```
}
```

```
int main(int argc, constchar * argv[ ]) {
```



```
int A[] = {1, 2, 3, 4, 5, 6, 7};
InsertionSortedArray(A, 7, 6);
Traverse(A, 7);

}
```

2. Finding the number which is not repeated in Array of integers, others are present for two times.

ALGORITHM: Arr_func(A[], N)

BEGIN:

```
K ← 0, c, B[]
FOR i ← 0 TO N DO
  c ← 0
  FOR j ← 0 TO N DO
    IF A[j] == A[i] THEN
      C ← c+1
  IF c == 1 THEN
    B[k++] ← A[i]
  FOR i ← 0 TO k DO
    WRITE(B[i])
END;
```

Time Complexity: $\Theta(N^2)$

The inner loop executes a statement N times. The outer loop executes the inner loop n times and conditional statement. The second loop (from i to k) executes a single statement in the above case. Hence, the total number of statements will be $n*(n+1) + 1$

Space Complexity: $\Theta(1)$

Three extra variables are used k, c & B[]. In case of a single nonrepetitive element, B will be a single element array.

```
#include <stdio.h>
void Arr_func(int A[], int N){
  int k=0, c, B[100];
  int i, j;
```

```
for (i=0; i <= N; i++) {
    c=0;
    for (j=0; j <= N; j++) {
        if (A[j] == A[i]) {
            c=c+1;
        }
    }
    if (c == 1) {
        B[k++]=A[i];
    }
}
for (i=0; i <= k; i++) {
    printf("%d", B[i]);
}
}
```

```
int main(int argc, constchar * argv[ ]) {
// Insert code here...
int A[ ]={4, 6, 3, 6, 3, 2, 6, 8, 1, 0, 3, 1, 4, 7, 4};
Arr_func(A, 15);
return0;
}
```

4.4. INDEX FORMULA

Given an Array and its base address. If we compute the address of the desired element through the formula, it is called the index formula of the array. This can be used while writing program for an array using pointers.

4.4.1. One Dimensional Array

Consider a one dimensional Array $A[L:U]$. Here L is the smallest index of the array and U is the highest index of the array. If we want to find out the number of elements in this array, it can be computed as $U-L+1$. e.g., if the Array is $A[0:9]$ as in C Language, no of elements are $9-0+1=10$.

For computing the Index formula we will first assume that the indexes are starting from 1 and each element occupies 1 Byte for storage. In the end, these assumptions will be removed.

Address of $A[1] = \alpha$

Address of $A[2] = \alpha + 1$

Address of $A[3] = \alpha + 2$

...

Address of $A[i] = \alpha + (i-1)$

Removing assumptions

If each byte takes N bytes for storage

*Address of $A[i] = \alpha + (i-1) * N$*

If Array indexes are starting from L , index i will be $i-L+1$

*Address of $A[i] = \alpha + (i - L + 1 - 1) * N$*

$\text{Address of } A[i] = \alpha + (i - L) * N$
--

4.4.2. Two Dimensional Array

Consider A two dimensional Array, e.g., $A[L_1:U_1][L_2, U_2]$. Array elements can be thought of being stored as Row Major Order fashion (First elements of first row are stored in the memory followed by second row and so on) or Column Major Order fashion (First elements of first column are stored in the memory followed by second Column and so on)

4.4.2.1. Row Major Order

In array $A[L_1:U_1][L_2, U_2]$, number of rows = $U_1 - L_1 + 1$ & number of columns = $U_2 - L_2 + 1$.

$\begin{aligned} \text{Address of } A[1][1] &= \alpha \\ \text{Address of } A[1][2] &= \alpha + 1 \\ \text{Address of } A[1][3] &= \alpha + 2 \\ \text{Address of } A[1][U_2] &= \alpha + U_2 - 1 \\ \text{Address of } A[2][1] &= \alpha + U_2 \\ \text{Address of } A[3][1] &= \alpha + 2 * U_2 \\ \text{Address of } A[4][1] &= \alpha + 3 * U_2 \\ &\dots \\ \text{Address of } A[i][1] &= \alpha + (i-1) * U_2 \\ \text{Address of } A[i][2] &= \alpha + (i-1) * U_2 + 1 \\ \text{Address of } A[i][3] &= \alpha + (i-1) * U_2 + 2 \\ &\dots \\ \text{Address of } A[i][j] &= \alpha + (i-1) * U_2 + (j-1) \end{aligned}$
--

For computing the Index formula, we will first assume that the indexes are starting from 1 and each element occupies 1 Byte for storage. In the end, these assumptions will be removed.

	1	2	3	...	j	...	U_2
1							
2							
3							
...							
i							
...							
U_1							

Removing assumptions, if every element requires N byte for storage, and row indexes are starting from L_1 and Column indexes are starting from L_2 .

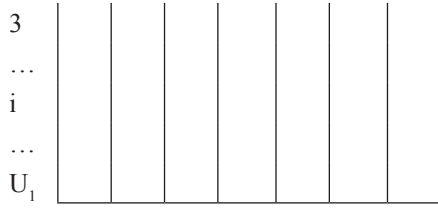
$$\text{Address of } A[i][j] = \alpha + [(i-L_1+1-1)*(U_2-L_2+1) + (j-L_2+1-1)] * N$$

$$\text{Address of } A[i][j] = \alpha + [(i-L_1)*(U_2-L_2+1) + (j-L_2)] * N$$

4.4.2.2. Column Major Order

Address of $A[1][1] = \alpha$
 Address of $A[2][1] = \alpha + 1$
 Address of $A[3][1] = \alpha + 2$
 Address of $A[U_1][1] = \alpha + U_1 - 1$
 Address of $A[1][2] = \alpha + U_1$
 Address of $A[1][3] = \alpha + 2*U_1$
 Address of $A[1][4] = \alpha + 3*U_1$
 ...
 Address of $A[1][j] = \alpha + (j-1)*U_1$
 Address of $A[2][j] = \alpha + (j-1)*U_1 + 1$
 Address of $A[3][j] = \alpha + (j-1)*U_1 + 2$
 ...
 Address of $A[i][j] = \alpha + (j-1)*U_1 + (i-1)$

	1	2	3	...	j	...	U_2
1							
2							



Removing assumptions, if every element requires N byte for storage, and row indexes are starting from L_1 and Column indexes are starting from L_2

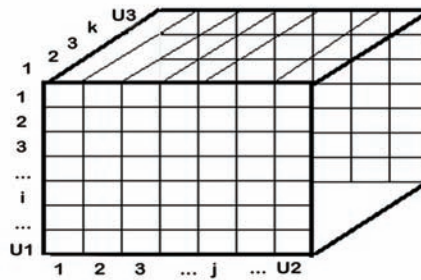
$$\text{Address of } A[i][j] = \alpha + [(j-L_2+1-1)*(U_1-L_1+1) + (i-L_1+1-1)] * N$$

$$\text{Address of } A[i][j] = \alpha + [(j-L_2)*(U_1-L_1+1) + (i-L_1)] * N$$

4.4.3. Three Dimensional Array

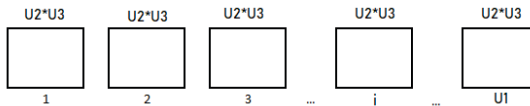
Consider A two dimensional Array, e.g., $A[L_1:U_1][L_2, U_2][L_3, U_3]$

A 3-D array can be visualized as a cuboid.



4.4.3.1. Row Major Order

If the slices are cut horizontally, we will get the slices of size $U_2 * U_3$. There will be U_1 such slices



$$\text{Address of } A[1][1][1] = \alpha$$

$$\text{Address of } A[2][1][1] = \alpha + (U_2 * U_3)$$

$$\text{Address of } A[3][1][1] = \alpha + (U_2 * U_3) * 2$$

$$\text{Address of } A[4][1][1] = \alpha + (U_2 * U_3) * 3$$

...

Address of $A[i][1][1] = \alpha + (U_2 * U_3) * (i-1)$

i^{th} row is a 2 Dimensional array of size $(U_2 * U_3)$

Address of $A[i][2][1] = \alpha + (U_2 * U_3) * (i-1) + (U_3)$
 Address of $A[i][3][1] = \alpha + (U_2 * U_3) * (i-1) + (U_3) * 2$
 Address of $A[i][4][1] = \alpha + (U_2 * U_3) * (i-1) + (U_3) * 3$
 ...
 Address of $A[i][j][1] = \alpha + (U_2 * U_3) * (i-1) + (U_3) * (j-1)$
 Address of $A[i][j][2] = \alpha + (U_2 * U_3) * (i-1) + (U_3) * (j-1) + 1$
 Address of $A[i][j][3] = \alpha + (U_2 * U_3) * (i-1) + (U_3) * (j-1) + 2$
 Address of $A[i][j][4] = \alpha + (U_2 * U_3) * (i-1) + (U_3) * (j-1) + 3$
 ...
 Address of $A[i][j][k] = \alpha + (U_2 * U_3) * (i-1) + (U_3) * (j-1) + (k-1)$

1							
2							
3							
...							
j							
...							
U_2							
	1	2	3	...	k	...	U_3

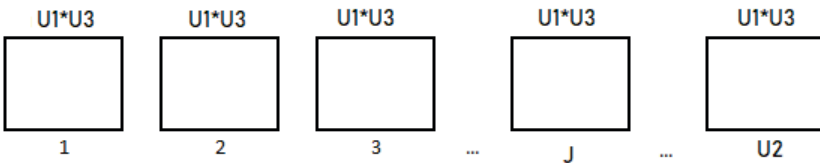
Removing Assumptions:

Address of $A[i][j][k] = \alpha + [((U_2 - L_2 + 1) * (U_3 - L_3 + 1)) * (i - L_1 + 1 - 1) + (U_3 - L_3 + 1) * (j - L_2 + 1 - 1) + (k - L_3 + 1 - 1)] * N$

Address of $A[i][j][k] = \alpha + [((U_2 - L_2 + 1) * (U_3 - L_3 + 1)) * (i - L_1) + (U_3 - L_3 + 1) * (j - L_2) + (k - L_3)] * N$

4.4.3.2. Column Major Order

If the slices are cut vertically, we will get the slices of size $U_1 * U_3$. There will be U_2 such slices



Address of $A[1][1][1] = \alpha$

Address of $A[1][2][1] = \alpha + (U_1 * U_3)$

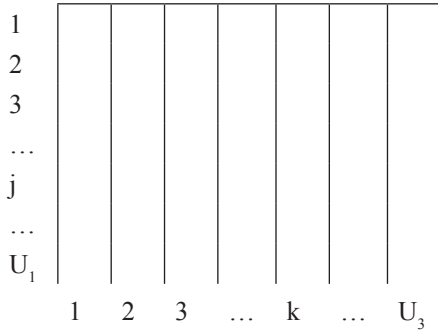
Address of $A[1][3][1] = \alpha + (U_1 * U_3) * 2$

Address of $A[1][4][1] = \alpha + (U_1 * U_3) * 3$

...

Address of $A[1][j][1] = \alpha + (U_1 * U_3) * (j-1)$

j^{th} column is a 2 Dimensional array of size $(U_1 * U_3)$



Address of $A[1][j][2] = \alpha + (U_1 * U_3) * (j-1) + (U_1)$

Address of $A[1][j][3] = \alpha + (U_1 * U_3) * (j-1) + (U_1) * 2$

Address of $A[1][j][4] = \alpha + (U_1 * U_3) * (j-1) + (U_1) * 3$

...

Address of $A[1][j][k] = \alpha + (U_1 * U_3) * (j-1) + (U_1) * (k-1)$

Address of $A[2][j][2] = \alpha + (U_1 * U_3) * (j-1) + (U_1) * (k-1) + 1$

Address of $A[3][j][3] = \alpha + (U_1 * U_3) * (j-1) + (U_1) * (k-1) + 2$

Address of $A[4][j][4] = \alpha + (U_1 * U_3) * (j-1) + (U_1) * (k-1) + 3$

...

Address of $A[i][j][k] = \alpha + (U_1 * U_3) * (j-1) + (U_1) * (k-1) + (i-1)$

Removing Assumptions:

Address of $A[i][j][k] = \alpha + [((U_1 - L_1 + 1) * (U_3 - L_3 + 1)) * (j - L_2 + 1 - 1) + (U_1 - L_1 + 1) * (k - L_3 + 1 - 1) + (i - L_1 + 1 - 1)] * N$

Address of $A[i][j][k] = \alpha + [((U_1 - L_1 + 1) * (U_3 - L_3 + 1)) * (j - L_2) + (U_1 - L_1 + 1) * (k - L_3) + (i - L_1)] * N$

EXERCISES

1. Given an array $A[lb:ub]$. The number of elements in this array is given by the formula
 - a. $ub - lb$
 - b. $ub - lb + 1$
 - c. $ub - lb - 1$
 - d. $lb - ub + 1$
2. Given an array $A[-1:7, 0:6]$. What is the number of rows and columns in this array
 - e. 9, 7
 - f. 7, 6
 - g. 8, 6
 - h. None of these
3. There are 10 elements in an array. If 4th element is deleted, position of how many elements is changed?
 - i. 7
 - j. 8
 - k. 6
 - l. 5
4. Find the Address of Element $A[4][6][9]$ in a row-major order array $A[0:10][-1:8][0:12]$. Consider each element required 4 bytes for storage and base address be 12000.
5. The address of element $P[6, 12]$ in the row major order array $P[-1:10, -2:20]$ where every element takes 2 bytes of storage with base address 2000 is
6. Find the Address of Element $A[3][8]$ in a row-major order array $A[-2:10][0:11]$. Consider each element required 8 bytes for storage and base address be 5000.
6. No. of elements in the array $A[-2:5, -1:5, 0:5]$ is
 - a. 48
 - b. 125
 - c. 336
 - d. 512

Programming Question

1. Given an unsorted array, find the length of the longest sequence of consecutive numbers in the array
2. Given k sorted arrays, Write a Program to merge them into a single sorted array.
3. Given an array of integers where each value $1 \leq x \leq \text{len}(\text{array})$, write a function that finds all the duplicates in the array. Click for the solution.

5

Searching Algorithms

CONTENTS

5.1. What Is Searching?.....	46
5.2. Sequential Search (Linear Search)	46
5.3. Binary Search	48
5.4. Comparison Of Linear And Binary Search.....	51
5.5. Indexed Sequential Search.....	52
Exercises.....	56

5.1. WHAT IS SEARCHING?

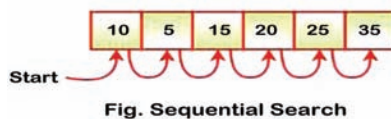
Consider a telephone directory of your mobile phone in which a telephone number is required to be searched through name. A dictionary where meaning is required to be searched for a given word. A catalogue from which description is required to be searched for a particular item. Daily attendance through finger print scan (biometric search). Any database search for record corresponding to some key e.g., Roll No. A keyword search in Google. Name search in etc. Searching is the process of finding a given value position in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of finding a particular item in a collection of items.

5.1.1. Searching Techniques

- Sequential search
- Binary search
- Index sequential search
- Tree based search

5.2. SEQUENTIAL SEARCH (LINEAR SEARCH)

- Sequential search is also called as linear search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a naive (brute force approach) searching algorithm.
- Sequential search compares the search key with all the elements in the list. If the element is matched, it returns the index, else it returns -1.



The above figure shows how sequential search works. It searches a key in an array till the key element is not found. If we search the element 25, it will go step by step in a sequence order. Sequential search is applied on the unordered list.

ALGORITHM `Linear_Search(A[], N, key)`

BEGIN:

FOR $i \leftarrow 1$ TO N DO

IF $A[i] == \text{key}$ THEN

RETURN i

RETURN -1

END;

Worst Case Time Complexity: $O(N)$

If the key element is found at the last index or not found, $N+1$ statements will be executed

Best Case Time Complexity: (1)

If the key element is found at the first index, only one statement will be executed

Space Complexity: $\theta(1)$

Only one extra variable is taken namely i

C Program

```
/******  
#include <stdio.h>  
/******  
int Linear_search(int A[ ], int N, int key){  
    int i=1;  
    for (i=1; i <= N; i++) {  
        if (A[i] == key) {  
            return i;  
        }  
    }  
    return -1;  
}  
/******  
int main(int argc, constchar * argv[ ]) {  
    int A[ ]={1, 2, 3, 4, 5, 6, 7, 8, 9};  
  
    int position=Linear_search(A, 9, 7);  
    printf("%d", position);  
    return 0;  
}
```

5.3. BINARY SEARCH

Binary Search is performed on a sorted array. It is a fast search algorithm with run-time complexity of $O(\log_2 n)$. Binary search works on the principle of divide and conquer. This searching technique looks for a particular element by comparing the median element of the collection. It is useful when there are large number of elements in an array.

- In Binary search, Mid index is calculated with the formula $\text{Mid} = (\text{Low} + \text{High})/2$. Here Low and High denote the smallest and largest index of the array. Search key is compared with the Mid Element of the array. If there is a match, search is successful, return Mid Index and stop the search process.
- In case search key is less than Mid Element of the array, it is likely that search key will be there in the Left part of the array. Hence High is changed to Mid-1 keeping Low index same.
- In case search key is larger than Mid Element of the array, it is likely that search key will be there in the Right part of the array. Hence Low is changed to Mid+1 keeping High index same.
- Whether search is performed on Left or Right part of the array, array size (search area reduces to half than previous).
- The entire process is repeated until either search is successful or we come to conclusion that element can not be there in the array.
- $\text{Low} < \text{High}$ means there are at least 2 elements in the array. $\text{Low} = \text{High}$ means there is exactly 1 element in the array. In case $\text{Low} > \text{High}$ is the terminating condition of the algorithm.

Example:

Consider the array having 5 elements and Search Key be 11. Binary search process goes like:

Low				high
[1]	[2]	[3]	[4]	[5]
10	11	13	18	20

(i)

$$\text{Mid} = (1+5)/2 = 3$$

$$A[3] \neq 11$$

since $11 < A[3]$ search will be done in Left half keeping the low same and $\text{high} = \text{mid}-1$

$$\text{high} = 3-1=2$$

(ii)

low	High
[1]	[2]
10	11

$\text{Mid} = (1+2)/2 = 1$

$A[1] \neq 11$

since $11 > A[1]$ search will be done in Right half keeping the high same and $\text{low} = \text{mid} + 1$

$\text{mid} = 1 + 1 = 2$

(iii)

Low, High
[1]
10

$\text{Mid} = (2+2)/2 = 2$

Since $11 = A[2]$, search completes. 2 will be returned

ALGORITHM Binary_search(A[], N, key)

BEGIN:

LOW $\leftarrow$ 1

HIGH $\leftarrow$ N

WHILE LOW $\leq$ HIGH DO

MID $\leftarrow$ (LOW+HIGH)/2

IF $A[\text{MID}] == \text{key}$ THEN

RETURN MID

ELSE

IF $\text{key} < A[\text{MID}]$ THEN

HIGH $\leftarrow$ MID-1

ELSE

LOW $\leftarrow$ MID+1

RETURN -1

END;

Time Complexity: $O(\log_2 N)$, $\Omega(1)$

If the element is found at the mid of the array in first distribution itself, three statements are executed (Best Case)

The above algorithm breaks the array into half in each iteration.

We divide it by 2 until we have only one element-

Time Complexity can be written as

$$T(n) = T(n/2) + C$$

Here C represents a constant number of statements for dividing the array to half size and deciding in which part of the array searching should be done.

$$T(n) = T(n/2^2) + C + C$$

$$= T(n/2^2) + 2 * C$$

$$= T(n/2^3) + 3 * C$$

...

$$= T(n/2^K) + K * C$$

Suppose array size reduce to 1 in k^{th} division, hence $n/(2^k)=1$

It can be rewritten as -

$$2^k = n$$

by taking log both side, we get

$$k = \log_2 n$$

$$T(n) = T(n/2^K) + K * C$$

$$= T(1) + C * \log_2 n$$

$$= 1 + C * \log_2 n$$

$$= O(\log_2 n)$$

Space Complexity: $\Theta(1)$

Three variables are used, i.e., HIGH, MID, and LOW

C PROGRAM

```
/******
```

```
#include <stdio.h>
```

```
/******
```

```
int BinarySearch(int A[ ], int N, int key){
```

```
int high=N-1, low=0;
```

```

while (low <= high) {
int mid=(low+high)/2;
if (A[mid] == key) {
return mid;
}
else{
if (key<A[mid]) {
high=mid-1;
}
else{
low=mid-1;
}
}
}
return -1;
}
/*****/

int main(int argc, constchar * argv[ ]) {
int A[]={1, 2, 3, 4, 5, 6, 7, 8, 9};
int position=BinarySearch(A, 9, 3);
printf("%d", position);
return 0;
}
/*****/

```

5.4. COMPARISON OF LINEAR AND BINARY SEARCH

The table given below compares the worst-case behavior of Linear and Binary Search

	Worst Case Time Complexity		
Elements	Linear Search	Binary Search	Explanation
1024	1024	10	1024 is 2^{10} , $\log_2 2^{10}$ is 10
1024*1024	1024*1024	20	1024*1024 is 2^{20} , $\log_2 2^{20}$ is 20
1024*1024*1024	1024*1024*1024	30	1024*1024*1024 is 2^{30} , $\log_2 2^{30}$ is 30

➔ It is quite evident from above table that Binary search is a far better technique than Linear Search.

5.5. INDEXED SEQUENTIAL SEARCH

In this searching method, an index is created, that contains some specific group or division of required record. From the index, the lower and upper limit of the range is found where the search key is expected to lie. The sequential search is performed in the found range.

Characteristics of indexed sequential search:

- Index sequential is performed on sorted list.
- Index is created by selecting equidistant (in terms of indexes) elements of the original array.
- The index is searched first then the array and guides the search in the array.

ALGORITHM: INDsearch(A[], N, KEY)

BEGIN:

Ind[], b[]

N1=0, c, d

FOR i ← 0 TO N STEP-2 DO

Ind[n1] ← i

b[n1] ← i

n1 ← n1+1

IF KEY<Ind[0] THEN

WRITE("NO. NOT FOUND")

ELSE

FOR i ← 1 TO n1 DO

IF KEY<Ind[i] THEN

c ← b[i-1]

d ← b[i]

BREAK

FOR i ← c TO d+1 DO

IF KEY == a[i] THEN

j ← 1

BREAK

IF j == 1 THEN

```

WRITE(i+1)
ELSE
WRITE("No. not found")
END;

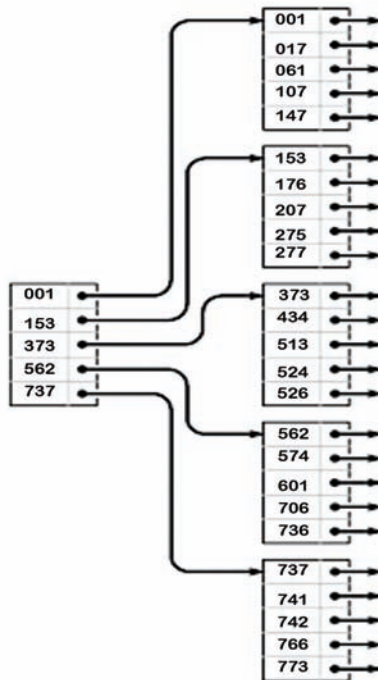
```

Time Complexity: $\Theta(N/K)$

If index is created by selecting each k^{th} element in the list, size of index is N/K upon which sequential search is performed.

Space Complexity: $\Theta(N/K)$

If index is created by selecting each k^{th} element in the list, size of index is N/K . This is the additional space than the original array. In case Index array is given, Space complexity will be $\Theta(1)$



C-PROGRAM

```

/*****
#include <stdio.h>
#include <stdlib.h>
*****/

```

```
void INDsearch(int A[ ], int n, int key)
{
int elements[20], indices[20], temporary, i;
int j = 0, ind = 0, start, end;
for (i = 0; i < n; i += 3) {
    elements[ind] = A[i];
    indices[ind] = i;
    ind++;
}
```

```
if (key < elements[0]) {
printf("Not found");
exit(0);
}
else {
for (i = 1; i <= ind; i++)
if (key < elements[i]) {
    start = indices[i-1];
end = indices[i];
break;
}
for (i = start; i <= end; i++) {
if (key == A[i]) {
    j = 1;
break;
}
}
if (j == 1)
printf("Found at index %d", i);
else
printf("Not found");
}
```

```
/******
```

```
void main()
{

int arr[ ] = { 10, 5, 16, 14, 20 };
int n =5;
int k = 8;
INDsearch(arr, n, k);
}
/*****/
```

EXERCISES

1. Rewrite the index sequential search algorithm with performing the binary search on index.
2. Write the Index Sequential search considering the names stored as keys. The index are to be maintained with each alphabets.
3. Write a C Program for performing a search on the Table of student data. The seach may be performed on both the Name and Roll Number. Result of the search is the entire record related to student

Multiple Choice Questions

1. Which of the following is correct recurrence for worst case of Binary Search?
 - a. $T(n) = 2T(n/2) + O(1)$ and $T(1) = T(0) = O(1)$
 - b. $T(n) = T(n-1) + O(1)$ and $T(1) = T(0) = O(1)$
 - c. $T(n) = T(n/2) + O(1)$ and $T(1) = T(0) = O(1)$
 - d. $T(n) = T(n-2) + O(1)$ and $T(1) = T(0) = O(1)$
2. The average number of key comparisons done in a successful sequential search in a list of length it is:
 - a. $\log n$
 - b. $(n-1)/2$
 - c. $n/2$
 - d. $(n+1)/2$
3. The average case analysis of Linear search on the array of size n requires comparison
 - a. $n/2$
 - b. n
 - c. $(n-1)/2$
 - d. $(n+1)/2$

6

Sorting Algorithms

CONTENTS

6.1. Bubble Sort.....	58
6.2. Insertion Sort	60
6.3. Selection Sort.....	64
6.4. Quick Sort	67
6.5. Merge Sort.....	74
6.6. Set Operations Using Array	80
6.7. Counting Sort.....	91
6.8. Radix Sort.....	94
Exercises	95

Sorting is a method of arranging the elements in the desired sequence. This sequence is expected to be either ascending or descending. In the current chapter, elements are organized in ascending sequence.

6.1. BUBBLE SORT

If some detergent is mixed in the bucket of water, the bubbles are created. Larger volume bubble has the tendency of reaching on the surface faster than smaller volume bubbles. In Bubble Sort Algorithm largest element is fixed in the first iteration, followed by the second largest element and so on so forth. In the process, the 1st element is compared with 2nd, and if the previous element is larger than the next one, elements are exchanged with each other. Then 2nd and 3rd elements are compared, and if second is larger than the 3rd, they are exchanged. The process repeats and finally (n-1)st element gets compared with nth and if previous one is larger than the next one, they are exchanged. This completes the first iteration. As a result largest element occupies its appropriate position in the sorted array, i.e., last position.

In the next iteration, the same process is repeated but one less comparison is performed than previous iteration (as largest number is already in place). As a result of second iteration, second largest number reaches to its appropriate position in the sorted array.

The similar iterations are repeated n-1 times. The result is the sorted array of n elements.

Below given figure shows how Bubble Sort works:

ALGORITHM: BubbleSort(A[], N)

BEGIN:

FOR i ← 0 TO N-1 DO

FOR j ← 0 TO N-i-1 DO

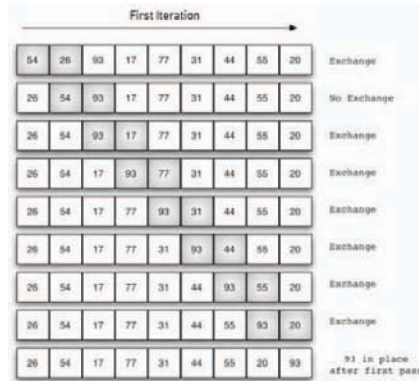
IF A[j]>A[j+1]

k ← A[j]

A[j] ← A[j+1]

A[j+1] ← k

END;



Above diagram shows the first iteration of Bubble sort. Largest no reaches to the last index. In second iteration second largest element will reach the second last index. The process is repeated for $n-1$ iterations.

$n-1$ comparisons in first iteration

$n-2$ comparison in second iteration

$n-3$ comparison in third iteration

...

1 Comparison in $(n-1)^{\text{st}}$ iteration

Total comparisons

$$= (n-1) + (n-2) + (n-3) + \dots + 1$$

$$= n(n-1)/2$$

$$= n^2/2 - n/2$$

If exchanges takes place with each comparison, Total number of statements to be executed are

$(n^2/2 - n/2) * 3$ as 3 statements are required for exchange.

Time Complexity: $\Theta(N^2)$

It will take $n-1$ outer iterations to sort the whole array unconditionally. $n-1$ comparisons in first iteration along with reducing number of comparisons in subsequent iterations.

Space Complexity: $\Theta(1)$

Only two variables are taken in any case, i.e., i , and j . Exchange may cause one more extra variable.

Optimized Bubble Sort

The Bubble sort can be stopped at the iteration in which no exchanges are observed as there will be no further exchanges in the subsequent iterations.

ALGORITHM: OptimizedBubbleSort(A[], N)**BEGIN:**FOR i $\leftarrow$ 0 TO N-1 AND Flag == 1 DOflag $\leftarrow$ 0FOR j $\leftarrow$ 0 TO N-i-1 DO

IF A[j]>A[j+1]

Flag $\leftarrow$ 1k $\leftarrow$ A[j]A[j] $\leftarrow$ A[j+1]A[j+1] $\leftarrow$ k**END;****Best Case Time Complexity (N)**

When the array is already sorted, the iteration will stop after comparing all the elements in the first iteration only making $n-1$ comparisons only and Best Case appears.

Worst Case Time Complexity $O(N^2)$

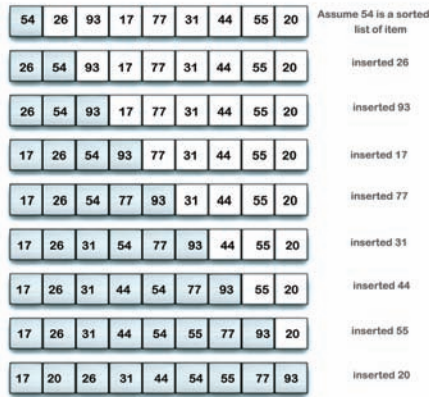
When the array is reverse sorted, it will take $n-1$ outer iterations to sort the whole array

Space Complexity: $\Theta(1)$

Three variables are taken in any case, i.e., i, j, and flag. Exchange may cause one more extra variable.

6.2. INSERTION SORT

- *Insertion sort* is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists. It considers two halves in the same array: Sorted and unsorted. To start with only one element is taken in the sorted list (first element) and $n-1$ elements in the unsorted list (2nd to nth element). It works by taking elements from the unsorted list one by one and inserting them in their correct position into the sorted list.



For writing the Algorithm, let us first write the algorithm for the first iteration only

In first pass

$j \leftarrow 2;$

$\text{Key} \leftarrow A[2]$

WHILE $\text{key} < A[j-1]$ AND $j > 1$ DO

$A[j] \leftarrow A[j-1];$

$j \leftarrow j-1$

$A[j+1] \leftarrow \text{Key};$

There will be $n-1$ similar iteration hence the above algorithm is taken into the outer loop that runs for $n-1$ times.

ALGORITHM: InsertionSort($A[], N$)

BEGIN:

FOR $i \leftarrow 2$ TO N DO

$\text{key} \leftarrow A[i]$

$j \leftarrow i-1$

WHILE $j > \leftarrow 1$ AND $A[j] > \text{key}$ DO

$A[j+1] \leftarrow A[j]$

$j \leftarrow j-1$

$A[j+1] \leftarrow \text{key}$

END;

If numbers are reverse sorted, each Insertion requires i shifts before placement (i is the iteration number).

1 shift in first iteration

2 shift in second iteration

3 shift in third iteration

...

$n-1$ shifts in $(n-1)^{\text{st}}$ iteration

Total shifts

$$= 1+2+3+ \dots +(n-1)$$

$$= n(n-1)/2$$

$$= n^2/2 - n/2$$

In each iteration placement of number is done once. Total placement = $n-1$

Looking at the above algorithm, there is one statement $\text{Key}=\text{A}[i]$ in each iteration and $j=j-1$ in the inner loop.

Total Statements to be executed

= Shifts + placement + loop decrement statement

$$= n^2/2 - n/2 + n-1 + (1+2+3+ \dots +n-1)$$

$$= n^2/2 - n/2 + n-1 + n^2/2 - n/2$$

$$= n^2-1$$

Worst Case Time Complexity: $O(n^2)$

If input numbers are reverse sorted, each Insertion requires i shifts before placement (i is the iteration number).

Best Case Time Complexity: (n)

If input numbers are sorted then no shifts are required in any iteration.

Total Statements to be executed

= Shifts+placement+loop decrement statement

$$= 0 + n-1 + 0$$

$$= n-1$$

Space Complexity: $\Theta(1)$

Only i & j and key variables are used in the above algorithm.

Related Task**1. Program for Sorting the given Complex Numbers****Algorithm:**

It is assumed that the Complex number is stored as a structure of the form
Struct Complex

```
{
int real;
int imaginary;
absolute;
};
```

$absolute = \sqrt{(real * real + imaginary * imaginary)}$ ($\sqrt{}$ is under root operator)

ALGORITHM SortComplexNumber(S[], N)**BEGIN:**

FOR $i \leftarrow 1$ TO $N-1$ DO

smallest $\leftarrow i$

FOR $j \leftarrow i+1$ TO N DO

IF $A[j].absolute < A[smallest].absolute$ THEN

$S \leftarrow j$

Exchange($A[smallest]$, $A[i]$)

END;**ALGORITHM Exchange($A[smallest]$, $A[i]$)****BEGIN:**

$treal \leftarrow A[smallest].real$;

$A[smallest].real \leftarrow A[i].real$;

$A[i].real \leftarrow treal$;

$timaginary \leftarrow A[smallest].imaginary$;

$A[smallest].imaginary \leftarrow A[i].imaginary$;

$A[i].imaginary \leftarrow timaginary$;

```

tabbsolute ← A[smallest].absolute;
A[smallest].absolute ← A[i].absolute;
A[i].absolute ← tabbsolute;
END;

```

Time Complexity: $\Theta(N^2)$

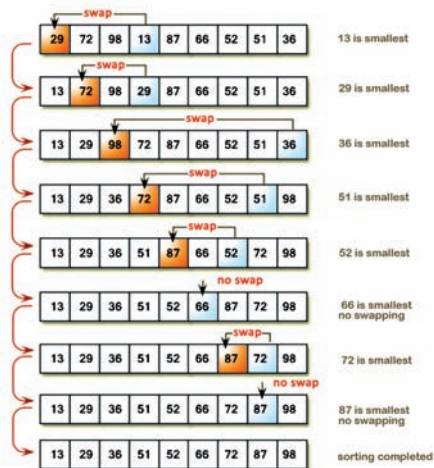
The two loops run the statements $n*n$ times, hence the complexity is N^2

Space Complexity: $\Theta(1)$

The 4 extra variables taken are i, j, t real & t imaginary in any case

6.3. SELECTION SORT

- The algorithm finds the minimum value, swaps it with the value in the first position in the first iteration. In the second iteration it finds minimum in the array from 2nd to nth position and swaps it with the value in the second position, and repeats these steps for the remainder of the list.
- It does no more than $n-1$ swaps, and thus is useful where swapping is very expensive.



$n-1$ comparisons in first iteration

$n-2$ comparison in second iteration

$n-3$ comparison in third iteration

...

1 Comparison in $(n-1)$ st iteration

Total = $(n-1) + (n-2) + (n-3) + \dots + 1$

$$= n(n-1)/2$$

$$= n^2/2 - n/2$$

Only one exchange is performed per iteration hence total $n-1$ exchanges are performed. Each exchange requires 3 statements.

Total statements for execution

$$= n^2/2 - n/2 + 3(n-1)$$

$$= n^2/2 + 5/2n - 3$$

ALGORITHM: SelectionSort(A[], N)

BEGIN:

FOR $i \leftarrow 1$ TO $N-1$ DO

$S \leftarrow i$

FOR $j \leftarrow i+1$ TO N DO

IF $A[j] < A[S]$ THEN

$S \leftarrow j$

Exchange($A[S]$, $A[i]$)

END;

Time Complexity: $\Theta(n^2)$

Since there are two unconditional nested loops, $n^2/2 + 5/2n - 3$ statements for execution. No best case as finding minimum in each iteration is compulsory.

Space Complexity: $\Theta(1)$

Only i & j variables are used in the above algorithm. Exchange may cause one more extra variable.

C-Program

```

/*****/
#include <stdio.h>
/*****/

void swap(int *xp, int *yp)
{
    int temporary = *xp;
    *xp = *yp;
    *yp = temporary;
}
/*****/

void SelectionSort(int A[ ], int n)

```

```

{
int i, j, min_idx;
for (i = 0; i < n-1; i++)
{
    min_idx = i;
    for (j = i+1; j < n; j++)
        if (A[j] < A[min_idx])
            min_idx = j;
    swap(&A[min_idx], &A[i]);
}
}
/*****/

void InsertionSort(int A[ ], int n)
{
int i, key, j;
for (i = 1; i < n; i++) {
    key = A[i];
    j = i-1;
    while (j >= 0 && A[j] > key) {
        A[j + 1] = A[j];
        j = j-1;
    }
    A[j + 1] = key;
}
}
/*****/

void BubbleSort(int A[ ], int n)
{
int i, j;
for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
        if (A[j] > A[j+1])

```

```

swap(&A[j], &A[j+1]);
}
/*****/

void Traverse(int A[ ], int n){
for (int i=1; i <= n; i++) {
printf("%d\t", A[i-1]);
}
}
/*****/

int main(int argc, constchar * argv[ ]) {
int A[ ]={5, 2, 3, 4, 9, 8, 7, 6, 1, 2};
int B[ ]={5, 2, 3, 4, 9, 8, 7, 6, 1, 2};
int C[ ]={5, 2, 3, 4, 9, 8, 7, 6, 1, 2};
BubbleSort(A, 10);
printf("\nbubble sort\n");
Traverse(A, 10);
InsertionSort(B, 10);
printf("Insertion Sort\n");
Traverse(B, 10);
SelectionSort(C, 10);
printf("Selection Sort\n");
Traverse(C, 10);
return 0;}
/*****/

```

6.4. QUICK SORT

Quicksort is a divide and conquer algorithm which relies on a *partition* operation. To partition an array, an element called a *pivot* is selected. All elements smaller than the pivot are moved before it, and all greater elements are moved after it. The Left and Right sublists are then recursively sorted using the same algorithm.

- Partition
- In partition, it is assumed that the last element is greater than all other elements in the array. In the first run to Partition algorithm, the last element is specifically taken larger than all other elements; however, in the subsequent run, the last element automatically becomes larger than

the set considered.

- Usually, the first element is picked as pivot. All other elements are compared with this element.
- Purpose of the Partition is to find the appropriate position of pivot in the sorted array.
- For the above purpose, smaller elements than pivot are kept on the Left part of the array and larger on the Right side.
- Once the appropriate position of pivot in the sorted array is found, it is returned and Quick Sort is called in Left and Right both halves.

ALGORITHM: Partition(A[], low, high)

BEGIN:

$i \leftarrow \text{low}$

$j \leftarrow \text{high}+1$

$\text{pivot} \leftarrow A[\text{low}]$

DO

DO

$i \leftarrow i+1$

WHILE($A[i] < \text{pivot}$)

DO

$j \leftarrow j-1$

WHILE($A[j] > \text{pivot}$)

IF $i < j$ THEN

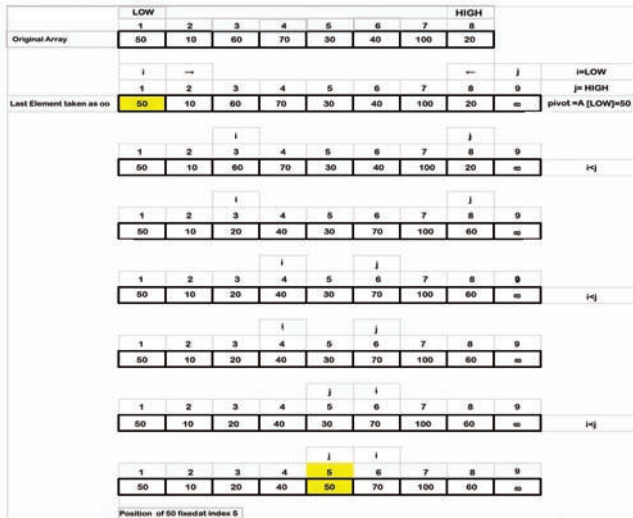
Exchange($A[i], A[j]$)

WHILE($i < j$)

Exchange($A[j], A[\text{low}]$)

RETURN j

END;



a. Number of Statements Execution

1. There are three fixed statements at the beginning before the loop.
2. i moves and stops at position where $A[i]$ is larger or equal than pivot element.
3. j moves and stops at position where $A[j]$ is smaller or equal than pivot element.
4. if $i < j$ then $A[i]$ and $A[j]$ are exchanged (smaller element on Left, larger on Right).
5. The process repeats until $i < j$. The moment i becomes greater than j, process stops and $A[j]$ and $A[Low]$ exchanged. j is the Right position of pivot.
6. j is returned

If i stops after each increment to it and j stops after each decrement to it and exchange of $A[i]$ and $A[j]$ is performed, total statements required to be executed for this operation will be:

Increment in i (1) + Decrement in j (1) + Exchange (3) = 5

Total $n/2$ such repeats will be done hence total statements = $5 * n/2$

At last, One exchange of $A[low]$ with $A[j]$ (3) + one return (1) = 4

Total Statements = $5 * n/2 + 4$

Complexity of Partition $\Theta(N)$

ALGORITHM: QuickSort($A[]$, low, high)

BEGIN:

IF low < high THEN

```

j ← Partition(A[ ], low, high)
QuickSort(A[ ], low, j-1)
QuickSort(A[ ], j+1, high)
END;

```

C PROGRAM

```

/*****/
#include <stdio.h>
/*****/

void Exchange(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
/*****/

int Partition(int A[ ], int low, int high){
    int i=low;int j=high+1;int pivot=A[low];
    do{
        do{
            i=i+1;
        } while (A[i]<pivot);

        do {
            j=j+1;
        } while (A[j]>pivot);
        if (i<j) {
            Exchange(&A[i], &A[j]);
        }
    } while(i<j);
    Exchange(&A[j], &A[low]);
    return j;
}
/*****/

```

```

void QuickSort(int A[ ], int low, int high){
if (low<high) {
int j=Partition(A, low, high);
QuickSort(A, low, j-1);
QuickSort(A, j+1, high);

}
}
/*****/

void Traverse(int A[ ], int n){
for (int i=0; i <= n-1; i++) {
printf("%d\t", A[i-1]);
}
}
/*****/

int main(int argc, constchar * argv[ ]) {
int A[11]={5, 2, 3, 4, 9, 8, 7, 6, 1, 2};
A[10]=32767;
QuickSort(A, 0, 9);
printf("\nQuick sort\n");
Traverse(A, 10);

return0;
}
/*****/

```

Let $T(n)$ represents the time complexity of Quick Sort. After partition two halves are created. Size of one half is $n/2$ and other one $n/2-1$. In case n is very large, $n/2-1$ can assumed to be very close to $n/2$. Recurrence for Quick Sort can be written as

$$T(n) = T(n/2) + T(n/2) + \theta(N)$$

$T(n/2)$ is the Time Complexity for Quick Sort on array of $n/2$ elements, $\theta(N)$ the time complexity of Partition

The recurrence can be re-written as:

$$T(n) = 2T(n/2) + \theta(N)$$

$$T(n) = 2T(n/2) + C N$$

$$\begin{aligned}
T(n) &= 2(2T(n/4) + C n/2) + C n \\
&= 2^2T(n/2^2) + 2Cn \\
&= 2^3T(n/2^3) + 3Cn \\
&\dots \\
&= 2^kT(n/2^k) + kCn
\end{aligned}$$

Assume that after k th division, the array size reduces to 1, i.e., $(n/2^k) = 1$

(assuming $2^k = n$, $\log_2 2^k = \log_2 n$, $k \log_2 2 = \log_2 n$, $k = \log_2 n$)

$$= n T(1) + C n \cdot \log_2 n$$

$= n \cdot 1 + C n \cdot \log_2 n$ (Effort to sort a problem of size 1 is 1, as Quicksort will not progress when low=high)

Out of n and $n \cdot \log_2 n$, $n \cdot \log_2 n$ is dominating, hence complexity will be written w.r.t $n \cdot \log_2 n$.

Best Case Time Complexity: ($n \log_2 n$)

It occurs when the elements are arranged in random order. The array breaks from the middle element:

$$T(n) = 2T(n/2) + \Theta(n)$$

Worst Case Time Complexity: $O(n^2)$

When the elements are already sorted, after each partition, first element is fixed, leaving no element in Left half and $n-1$ elements on Right half. Recurrence for this scenario can be written as:

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

When the Elements are reverse sorted, on time last element is fixed by Partition and next time the first element and it keeps alternating. In this case, also, the above recurrence is applicable.

$$T(n) = 1 + T(n-1) + C \cdot n$$

$T(0)$ is 1 as low>high in this case. It can be clubbed with $\Theta(N)$ as N is very larger

$$= T(n-1) + C \cdot n$$

$$= T(n-2) + C(n-1) + C n$$

$$= T(n-3) + C(n-2) + C(n-1) + C n$$

$$= T(1) + C \cdot 2 + \dots + C(n-2) + C(n-1) + C n$$

$$= 1 + C \cdot 2 + C \cdot 3 + \dots + C \cdot n$$

$$= 1 + C(2+3+4+\dots+n)$$

$$= 1 + C(n(n+1)/2 - 1)$$

$$= C \cdot n^2/2 + Cn - C + 1$$

Which can be written as $O(n^2)$

Space Complexity: $O(n)$, $\Omega(\log_2 n)$

In the worst case, the call stack formed by QuickSort can be of order n

In the best case, the array is split into two and until a single element is reached. At any point of time, the call stack doesn't exceed the $\log_2 n + 1$

6.4.1. Improving Worst Case

The major reason for the worst case to appear is the fixing of the extreme left or extreme right element by the partition. There are two ways by which this can be avoided.

1. Taking Median element as pivot

For this, Compute $\text{Mid} = (\text{Low} + \text{High})/2$ in the Partition and exchange the first element with the Mid element before actually going ahead with the partition

2. Taking random element as pivot

For this, randomly select an index in between Low and High in the Partition and exchange the first element with the element at randomly selected index before actually going ahead with the partition logic.

ALGORITHM: Partition(A[], low, high) // Randomized Quick Sort Partition
BEGIN:

$i \leftarrow \text{low}$

$j \leftarrow \text{high} + 1$

$r \leftarrow \text{Random}(\text{low}, \text{high})$

Exchange($A[\text{low}]$, $A[r]$)

pivot $\leftarrow A[\text{low}]$

DO

DO

$i \leftarrow i + 1$

WHILE($A[i] < \text{pivot}$)

DO

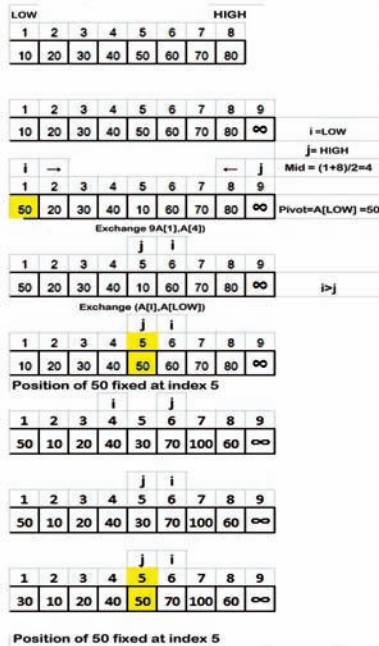
```

j ← j+1
WHILE(A[j]>pivot)

IF i<j THEN
Exchange(A[i], A[j])
WHILE(i<j)

Exchange(A[j], A[low])
RETURN j
END;

```

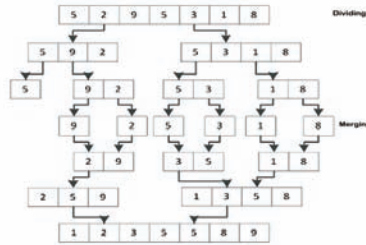


Both the above methods will reduce the possibility of selection of extreme Left or extreme Right element as the result of Partition.

6.5. MERGE SORT

- Merge Sort works on the principle of Divide and conquer.
- It keeps dividing the given array into two parts from very mid until we get an array of 1 element.

- Array of 1 element is supposed to be sorted.
- The two sorted arrays are merged to get the larger sorted array.
- The parts which are segregated from one can only merge.
- This is a recursive procedure.



To understand the Merge operation, first, try to understand the Merging of two sorted arrays.

Merging two sorted arrays

ALGORITHM: MergeArray(A[], m, B[], n) // m is the size of A[] and n is the size of B[] array

BEGIN:

C[m+n]

i ← 1, j ← 1, k ← 1

WHILE i ≤ m AND j ≤ n DO

//Comparing the elements of A[] and B[] arrays

IF A[i] < B[j] THEN

C[k] ← A[i]

i ← i+1

k ← k+1

ELSE

C[k] ← B[j]

j ← j+1

k ← k+1

WHILE i ≤ m DO

// If some of the elements of A[] array are Left out

C[k] ← A[i]

i ← i+1

k ← k+1

WHILE j ≤ n DO

// If some of the elements of B[] array

are Left out

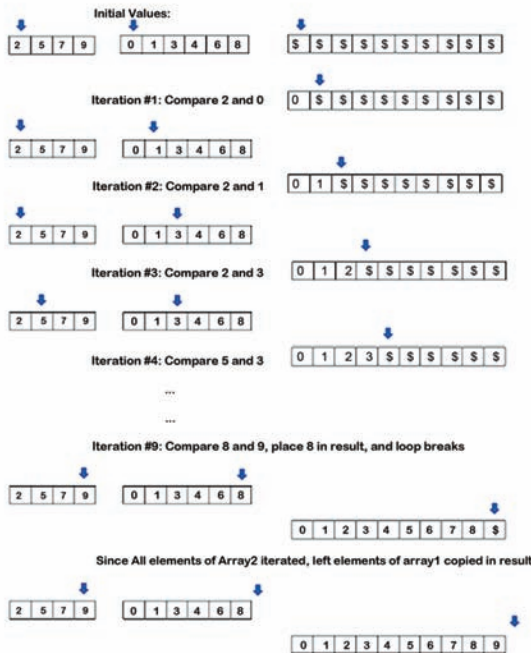
$C[k] \leftarrow B[j]$

$J \leftarrow j+1$

$k \leftarrow k+1$

RETURN C

END;



Time Complexity: $\Theta(m + n)$

For putting one element in the C array, one comparison is required. For $m+n$ elements, $m+n$ comparisons are required. Along with these three more statements, i.e., number placement in C array, increment in i/j and k . At last C array is returned.

Total statements = Comparison + placements + Return

$$= (m+n) + 3*(m+n) + 1$$

$$= 4*(m+n) + 1$$

Space Complexity: $\Theta(m+n)$

A new array of size $m+n$ is formed. Three extra variable i , j , and k are required. Total space $m+n+3$

In the case of Merge Sort, we require the sorted element in the input array itself. The original array is logically divided into two parts, which are sorted. After the

Merging in the temporary array, elements should be shifted back to the original array.

The first part of array has index from Low to Mid and second one from Mid+1 to High

ALGORITHM: Merge(A[], Low, Mid, High)

BEGIN:

$i \leftarrow \text{Low}, j \leftarrow \text{Mid}+1, k \leftarrow \text{High}$

WHILE $i \leq \text{Mid}$ AND $j \leq \text{High}$ DO

IF $A[i] < A[j]$ THEN

$C[k] \leftarrow A[i]$

$i \leftarrow i+1$

$k \leftarrow k+1$

ELSE

$C[k] \leftarrow A[j]$

$J \leftarrow j+1$

$k \leftarrow k+1$

WHILE $i \leq \text{Mid}$ DO

$C[k] \leftarrow A[i]$

$i \leftarrow i+1$

$k \leftarrow k+1$

WHILE $j \leq \text{High}$ DO

$C[k] \leftarrow A[j]$

$J \leftarrow j+1$

$k \leftarrow k+1$

FOR $i \leftarrow \text{low}$ TO high DO

$A[i] \leftarrow C[i]$

END;

In MergeArray procedure, size of the array is taken as m & n . Here size of arrays will be $n/2$ and $n/2$. Other than the statements in the MergeArray procedure, shifting of elements back to original array will require $n/2+n/2$ effort.

Total statements to be executed = Statements used in MergeArray + Shifting

$$= 4*(n/2+n/2) + 1+n/2+n/2$$

$$= 5*n+1$$

$$= \Theta(N)$$

ALGORITHM: MergeSort(A[], low, high)

BEGIN:

IF low<high DO

Mid $\leftarrow$ (low+high)/2

MergeSort(A[], low, mid)

MergeSort(A[], mid+1, high)

Merge(A, low, mid, high)

END;

C-Program

```
/******
```

```
#include <stdio.h>
```

```
*****
```

```
void Merge(int A[ ], int low, int mid, int high){
```

```
int i=low;int j=mid+1;int k=high;int C[k+1];
```

```
while ((i <= mid)&&(j <= high)) {
```

```
if (A[i]<A[j]) {
```

```
    C[k]=A[i];
```

```
    i=i+1;
```

```
    k=k+1;
```

```
}
```

```
else{
```

```
    C[k]=A[j];
```

```
    j=j+1;
```

```
    k=k+1;
```

```
}
```

```
while (i <= mid) {
```

```
    C[k]=A[i];
```

```
    i=i+1;
```

```
    k=k+1;
```

```
}
```

```

while (j <= high) {
    C[k]=A[j];
    j=j+1;
    k=k+1;
}
}
for (i=low; i <= high; i++) {
    A[i]=C[i];
}
}
/*****/

void MergeSort(int A[ ], int low, int high){
if (low<high) {
int mid=(low+high)/2;
MergeSort(A, low, mid);
MergeSort(A, mid+1, high);
Merge(A, low, mid, high);
}
}
/*****/

void Traverse(int A[ ], int n){
for (int i=1; i <= n; i++) {
printf("%d\t", A[i-1]);
}
}
/*****/

int main(int argc, constchar * argv[ ]) {
int A[ ]={5, 2, 3, 4, 9, 8, 7, 6, 1, 2};
MergeSort(A, 0, 9);
printf("\nMerge sort\n");
Traverse(A, 10);

return 0;
}

```

/******/

Merge Sort Process divides the original array in two parts of $n/2$ and $n/2$ size. Merging of both the parts requires $\Theta(n)$ effort.

Recurrence for the Merge sort can written as

$$T(n) = T(n/2) + T(n/2) + \Theta(n)$$

$T(n/2)$ is the Time Complexity for Merge Sort on array of $n/2$ elements, $\Theta(n)$ the time complexity of Merge Operation

The recurrence can be re-written as

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = 2T(n/2) + C n$$

$$T(n) = 2(2T(n/4) + C n/2) + C n$$

$$= 2^2T(n/2^2) + 2Cn$$

$$= 2^3T(n/2^3) + 3Cn$$

...

$$= 2^kT(n/2^k) + kCn$$

Assume that after k th division, the array size reduces to 1, i.e., $(n/2^k) = 1$

(assuming $2^k = n$, $\log_2 2^k = \log_2 n$, $k \log_2 2 = \log_2 n$, $k = \log_2 n$)

$$= n T(1) + C n \cdot \log_2 n$$

$= n \cdot 1 + C n \cdot \log_2 n$ (Effort to sort a problem of size 1 is 1 as Merge sort will progress only if the number of elements are 2 or more than 2, i.e., $\text{low} < \text{high}$. when $\text{low} = \text{high}$, there is only 1 element in the array and Merge sort will not progress)

Out of n and $n \cdot \log_2 n$, $n \cdot \log_2 n$ is dominating; hence complexity will be written w.r.t $n \cdot \log_2 n$.

Time Complexity: $\Theta(n \log_2 n)$

Notation Θ is used because there is no other case in this Algorithm.

Space Complexity: $\Theta(n)$

Here, a call stack is formed whose height is maximum $\log_2 n + 1$. But we are using an additional array of Size n in Merge. Since $n \gg \log_2(n)$, Space complexity is $\Theta(n)$

6.6. SET OPERATIONS USING ARRAY

1. Write a Program for Finding Set Elements of A That Belong to B.

ALGORITHM: A_AND_B(A[], m, B[], n)

BEGIN:

```

C[m+n]
i ← 1, j ← 1, k ← 1
WHILE i <= m AND j <= n DO
  IF A[i]<B[j] THEN
    i ← i+1
  ELSE
    IF A[i] == B[j] THEN
      C[k] ← B[j]
      i ← i+1
      j ← j+1
      k ← k+1
    ELSE
      j ← j+1
  RETURN C
END;
```

Time Complexity: $\Theta(m+n)$

The loop will continue until all the elements of A & B have been Traversed. Hence, it will take $m+n$ statements

Space Complexity: $\Theta(m+n)$

Here, a new array of size $m+n$ is formed.

C-Program

```

/*****/
#include <stdio.h>
/*****/

int C[1000];
int A_AND_B(int A[ ], int m, int B[ ], int n){

int i=1;int j=1;int k=1;
while ((i <= m)&&(j <= n)) {
  if (A[i]<B[j]) {
    i=i+1;
```

```

    }
elseif (A[i] == B[j]){
C[k]=B[j];
    i=i+1;
    j=j+1;
    k=k+1;
}
else{
    j=j+1;
}
}
return k;
}
/*****/

int main(int argc, constchar * argv[ ]) {
int A[ ]={1, 3, 4, 7, 9, 15};
int B[ ]={2, 3, 4, 8, 11};
int c=A_AND_B(A, 6, B, 5);
for (int i=1;i<c;i++) {
printf("%d\t", C[i]);
}
return 0;
}
/*****/

```

2. Write a Program for Finding Set Elements of A That Does Not Belong to B.

ALGORITHM: A_AND_NOT_B(A[], m, B[], n)

BEGIN:

C[m]

i ← 1, j ← 1, k ← 1

WHILE i ≤ m AND j ≤ n DO

IF A[i] < B[j] THEN

C[k] ← A[i]

k ← k+1

```
i ← i+1
ELSE
IF A[i] == B[j] THEN
i ← i+1
j ← j+1
ELSE
j ← j+1
WHILE i <= m DO
C[k] ← A[i]
i ← i+1
k ← k+1
RETURN C
END;
```

C-Program

```
/******
#include <stdio.h>
/******

int C[1000];
int A_AND_B(int A[ ], int m, int B[ ], int n){

int i=0;int j=0;int k=1;
while ((i <= m)&&(j <= n)) {
if (A[i]<B[j]) {
C[k]=A[i];
k=k+1;
i=i+1;
}
elseif (A[i] == B[j]){
i=i+1;
j=j+1;
}
else{
j=j+1;
```



```

    }
}
while (i <= m) {
C[k]=A[i];
    i=i+1;
    k=k+1;
}
return k;
}
/*****/

int main(int argc, constchar * argv[ ]) {
int A[ ]={1, 3, 4, 7, 9, 15};
int B[ ]={2, 3, 4, 8, 11};
int c=A_AND_B(A, 6, B, 5);
for (int i=1;i<(c-1);i++) {
printf(“%d\t”, C[i]);
}

return 0;
}
/*****/

```

Time Complexity: $\Theta(m+n)$

The loops will take $3*m$ statements to Traverse all the elements of A and compare them with the elements of B

Space Complexity: $\Theta(m)$

A new array of size m is formed to store the copied elements

3. Write a Program for Finding Set Union. Consider the Sorted Array as One Set.

ALGORITHM: SetUnion(A[], m, B[], n)**BEGIN:**

C[m+n]

i $\leftarrow$ 1, j $\leftarrow$ 1, k $\leftarrow$ 1

WHILE i $\leq$ m AND j $\leq$ n DO

```
IF A[i]<B[j] THEN
C[k] ← A[i]
i ← i+1
k ← k+1
ELSE
IF A[i] == B[j] THEN
C[k] ← B[j]
i ← i+1
j ← j+1
k ← k+1
ELSE
C[k] ← B[j]
j ← j+1
k ← k+1
WHILE i <= m DO
C[k] ← A[i]
i ← i+1
k ← k+1
WHILE j <= n DO
C[k] ← B[j]
j ← j+1
k ← k+1
RETURN C
END;
```

Time Complexity: $\Theta(m+n)$

The loop will continue until all the elements of A & B have been traversed. Hence, it will take $m+n$ statements.

Space Complexity: $\Theta(m+n)$

Here, a new array of size $m+n$ is formed.

C-Program

```
/*  
*****  
*/
```

```
#include <stdio.h>

/*****/

int C[1000];

int SetUnion(int A[ ], int m, int B[ ], int n){

    int i=0;int j=0;int k=1;
    while ((i<m)&&(j<n)) {
        if (A[i]<B[j]) {
            C[k]=A[i];
            k=k+1;
            i=i+1;
        }
        elseif (A[i] == B[j]){
            C[k]=B[j];
            i=i+1;
            j=j+1;
            k=k+1;
        }
        else{
            C[k]=B[j];
            j=j+1;
            k=k+1;
        }
    }
    while (i <= m) {
        C[k]=A[i];
        i=i+1;
        k=k+1;
    }
    while (j <= n) {
        C[k]=A[j];
        j=j+1;
        k=k+1;
    }
}
```

```

return k;
}
/*****/

int main(int argc, constchar * argv[ ]) {
int A[ ]={1, 3, 4, 7, 9, 15};
int B[ ]={2, 3, 4, 8, 11};
int c=SetUnion(A, 6, B, 5);
for (int i=1;i<=(c-2);i++) {
printf(“%d\t”, C[i]);
}

return 0;
}
/*****/

```

4. Write a Program for finding Set Intersection. Consider the sorted array as one set.

ALGORITHM: SetIntersection(A[], m, B[], n)

BEGIN:

C[m+n]

i ← 1, j ← 1, k ← 1

WHILE i ≤ m AND j ≤ n DO

 IF A[i] < B[j] THEN

 i ← i+1

 ELSE

 IF A[i] == B[j] THEN

 C[k] ← B[j]

 i ← i+1

 j ← j+1

 k ← k+1

 ELSE

 j ← j+1

RETURN C

END;

Time Complexity: $\Theta(m+n)$

The loop will continue until all the elements of A & B have been traversed. Hence, it will take $m+n$ statements

Space Complexity: $\Theta(m+n)$

Here, a new array of size $m+n$ is formed.

C-Program

```

/*****/
#include <stdio.h>
/*****/

int C[1000];

int SetIntersection(int A[ ], int m, int B[ ], int n){

    int i=0;int j=0;int k=1;
    while ((i<m)&&(j<n)) {
        if (A[i]<B[j]) {
            i=i+1;
        }
        elseif (A[i] == B[j]){
            C[k]=B[j];
            i=i+1;
            j=j+1;
            k=k+1;
        }
        else{
            j=j+1;
        }
    }
    return k;
}

/*****/

int main(int argc, constchar * argv[ ]) {
    int A[ ]={1, 3, 4, 7, 9, 15};

```

```

int B[ ]={2, 3, 4, 8, 11};
int c=SetIntersection(A, 6, B, 5);
for (int i=1;i<(c);i++) {
printf(“%d\t”, C[i]);
}
return 0;
}
/*****/

```

5. Write a Program for finding Set Difference. Consider the sorted array as one set.

ALGORITHM: SetDifference(A[], m, B[], n)

BEGIN:

C[m]

$i \leftarrow 1, j \leftarrow 1, k \leftarrow 1$

WHILE $i \leq m$ AND $j \leq n$ DO

 IF $A[i] < B[j]$ THEN

$i \leftarrow i+1$

 ELSE

 IF $A[i] = B[j]$ THEN

$i \leftarrow i+1$

$j \leftarrow j+1$

 ELSE

$C[k] \leftarrow B[j]$

$j \leftarrow j+1$

$k \leftarrow k+1$

WHILE $j \leq n$ DO

$C[k] \leftarrow B[j]$

$j \leftarrow j+1$

$k \leftarrow k+1$

RETURN C

END;

Time Complexity: $\Theta(m+n)$

The loop will continue until all the elements of A & B have been traversed. Hence, it will take $m+n$ statements.

Space Complexity: $\theta(m)$

Here, a new array of size m is formed.

C-Program

```

/*****/
#include <stdio.h>
/*****/
int C[1000];
int SetDifference(int A[ ], int m, int B[ ], int n){

int i=0;int j=0;int k=1;
while ((i<m)&&(j<n)) {
if (A[i]<B[j]) {
    i=i+1;
}
elseif (A[i] == B[j]){
    i=i+1;
    j=j+1;
}
else{
C[k]=B[j];
    j=j+1;
    k=k+1;
}
}
while (j <= n) {
C[k]=B[j];
    j=j+1;
    k=k+1;
}
return k;

```

```

}
/*****
int main(int argc, constchar * argv[ ]) {
int A[ ]={1, 3, 4, 7, 9, 15};
int B[ ]={2, 3, 4, 8, 11};
int c=SetDifference(A, 6, B, 5);
for (int i=1;i<=(c-1);i++) {
printf("%d\t", C[i]);
}

return0;
}
*****/

```

6.7. COUNTING SORT

Counting sort is an efficient algorithm for sorting an array of elements that each have a nonnegative integer keys. The largest number (k) is expected to be very less as compared to the size of array (n). Hence numbers will be repeated in the array. Usually, Counting Sort is not a direct technique. It is used as a base of Radix Sort.

This is the Stable Sort and relative ordering of the same numbers are maintained in the sorted array.

For implementation of Counting Sort, a counting array ($C[]$) is taken. It has a size equal to largest element in the array+1, i.e., $k+1$.

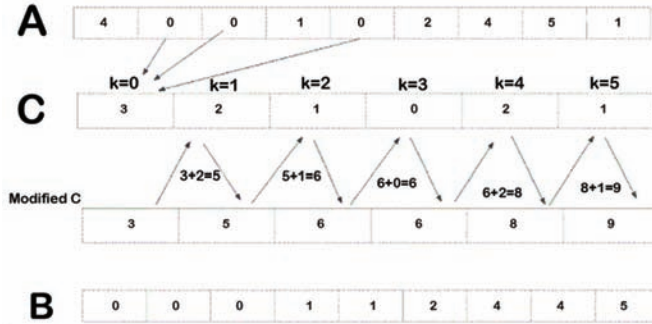
In each cell of C array, counting of element is maintained. $C[0]$ stores counting of 0s, $C[1]$ stores counting of 1s, $C[2]$ stores counting of 2s and so on so forth $C[k]$ stores counting of k s

After having count found, the C arrays is readjusted as following:

- $C[1]$ contains elements which are less than or equal to 0 an 1;
- $C[2]$ contains elements which are less than or equal to 1 an 2;
- $C[3]$ contains elements which are less than or equal to 2 an 3;
- ...
- $C[k]$ contains elements which are less than or equal to k an $k-1$.

For placement of numbers in B array, A array is scanned from back. The number is placed at the location in the C Array. e.g., if $A[i]$ is 7, $C[7]$ stores the index on B array in which $A[i]$ will be stored.

After placing the element in B array, $C[7]$ will be decremented by 1. This is done to ensure that next 7 will be stored at some other place in B array (prior to current 7). This way last 7 will be stored later in the B array than prior 7s. This way stability is maintained for all the elements.



ALGORITHM: CountingSort(A[], k, n)

BEGIN:

FOR $i \leftarrow 0$ TO k DO

$c[i] \leftarrow 0$

FOR $j \leftarrow 0$ TO n DO

$c[A[j]] \leftarrow c[A[j]] + 1$

FOR $i \leftarrow 1$ TO k DO

$c[i] \leftarrow c[i] + c[i-1]$

FOR $j \leftarrow n-1$ TO 0 STEP-1 DO

$B[c[A[j]]-1] \leftarrow A[j]$

$c[A[j]] \leftarrow c[A[j]] - 1$

RETURN B

END;

Time Complexity: $\Theta(n)$

The total number of statements executed are $2n + 2k$

Space Complexity: $\Theta(n)$

Two new arrays B[] of size n and C[] of size k are used

C-Program

```
/******
```

```
#include <stdio.h>
```

```
/******
```

```
void CountingSort(int A[ ], int b[ ], int N)
```

```
{
```

```
int K = 0;
```

```
for(int i=0; i<N; i++) {
```

```
if (A[i]>K) {
```

```
    K=A[i];
```

```
}
```

```
}
```

```
int C[K+1];
```

```
for(int i=0 ; i <= K; i++) {
```

```
    c[i] = 0;
```

```
}
```

```
for(int i=0; i<N; i++) {
```

```
    c[A[i]]++;
```

```
}
```

```
for(int i=1; i <= K; i++) {
```

```
    c[i]=c[i]+c[i-1];
```

```
}
```

```
for(int i=N; i >= 0; i--) {
```

```
    b[c[a[i]]-1]=a[i];
```

```
    c[a[i]]--;
```

```
}
```

```
}
```

```
/******
```

```
void Traverse(int A[ ], int n){
```

```
for (int i=1; i <= n; i++) {
```

```
printf("%d\t", A[i-1]);
```

```
}
```

```
}
```

```

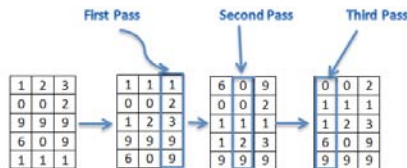
/*****/
int main(int argc, constchar * argv[ ]) {
int A[ ]={4, 5, 2, 3, 1, 7, 6, 8, 9, 1, 0, 3, 2, 1, 4, 3, 1, 6, 5, 1, 2, 3, 4, 5, 6, 7, 1, 2, 0,
5, 1, 0, 1, 2, 3, 4, 5, 0, 1, 5};
int B[40], C[10];
CountingSort(A, C, B, 40);
Traverse(B, 40);
return0;
}
/*****/

```

6.8. RADIX SORT

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value.

Usually, Radix sort is performed on the same length numbers. First Counting sort is performed on least significant digit (LSD) and numbers are REARranged. Counting Sort again is performed on each digit moving towards the Most significant digit.



ALGORITHM: RadixSort(A[], n, d)

BEGIN:

FOR i=1 TO d DO

 Apply counting Sort on A[] at radix i

END;

Time Complexity: $\Theta(d.n)$

Counting Sort is called for d times.

Space Complexity: $\Theta(n)$

Counting Sort is performed d times. Two new arrays B[] of size n and C[] of size k are used (as in Counting Sort)

EXERCISES

1. What is external Sort? Which among the studied algorithm in the chapter is External Sort?
2. Write an algorithm for Shell Sort
3. What is stable sort? Which among the sorting technique discussed in the chapter are stable?
4. Write Recursive procedure for Bubble Sort
5. Which among the sorting Algorithms discussed in the Chapter are the external Sorting?
6. Consider the array $A[] = \{6, 4, 8, 1, 3\}$ apply the insertion sort to sort the array. Consider the cost associated with each sort is 25 rupees, what is the total cost of the insertion sort when element 1 reaches the first position of the array?
7. An Array is given as $A = \{10, 30, 20, 50, 40\}$. If Bubble and Selection sorts are applied on Array A, which of these two will required less computations to sort this array?
8. An Array is given as $A = \{50, 20, 10, 40, 30, 60, 70, 90\}$. Explain the working of Merge Sort on the above array using Tree of Call of Merge Sort. How many activation records will pending at maximum there on stack at any moment for this example?
9. Pick the odd one out
 - a. Counting sort
 - b. Bucket sort
 - c. Shell Sort
 - d. Radix sort
10. The complexity of merge operation on two sorted arrays of size m & n (given $m > n$)
 - a. $O(mn)$
 - b. $O(m+n)$
 - c. $O(m/n)$
 - d. $O(m)$
11. Which item is selected in the first partition of Set <DATA STRUCTURES> by quick sort if pivot element is first element?
 - a. D
 - b. A
 - c. C
 - d. T
12. Match the elements of table1 with table2.

Table 1

Table 2

-
1. Linear search(worst case) a. $O(n \log n)$

- 2. Quick sort (worst case) b. $O(n)$
- 3. Binary Search (best case) c. $O(n^2)$
- 4. Merge Sort (average case) d. $O(1)$
- 13. Which among the following sorting technique is stable
 - a. Bubble
 - b. Selection
 - c. Counting
 - d. Merge
- 14. Which among the following algorithms does not require the sorted set as pre-requisite
 - a. Merge
 - b. Sequential search
 - c. Binary search
 - d. Index-Sequential search
- 15. Applying the quick sort algorithm on set $\langle 100, 50, 20, 120, 70, 30, 150, 10, 40 \rangle$, no of elements in left and right half after the first pass is:
- 16. Arranging the set of strings alphabetically is called

7

Heap & Heap Sort

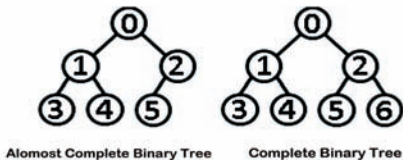
CONTENTS

7.1. Heap	98
7.2. Insertion In Heap	102
7.3. Deletion In Heap	103
7.4. Heap As Priority Queue	104
7.5. Heap Sort	105
Exercises	109

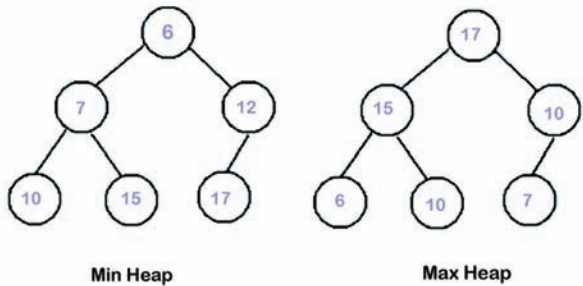
7.1. HEAP

Heap is a specialized tree-based data structure which satisfies two properties

1.
- Shape:** Shape of a binary tree is almost complete binary tree.



2.
- Order:** According to this there can be two types of heap:
- a.
- Min Heap:** Wherein the root node contains minimum key. All the parents in the tree contain less information than the children.
- b.
- Max Heap:** Wherein the root node contains Maximum key. All the parents in the tree contain more information than the children.

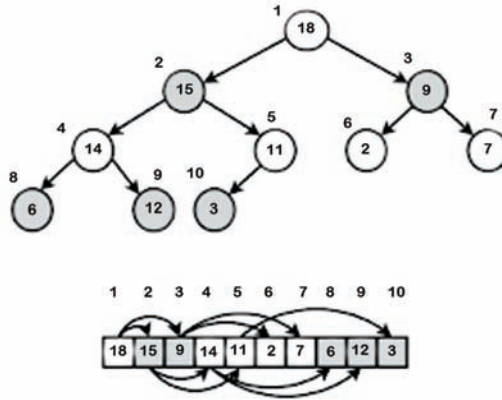


7.1.1. Creation of Heap

For constructing the heap, implicit Binary Tree Representation is chosen. According to this, if parent is at i index, Left child will be at $2*i$ index and Right child at $2*i+1$ index

If child is at some i index then parent will be at $i/2$ index.

Parent	Left Child	Right Child
I	$2*i$	$2*i+1$
Child	Parent	
I	$\text{Lower bound}(i/2)$	



To construct the heap, MaxHeapify or MinHeapify operation is performed. This operation converts each subtree to heap starting from bottom one and going up. A total of $n/2$ times adjustments are performed.

ALGORITHM MaxHeapify(A[], N)

BEGIN:

FOR $i \leftarrow N/2$ TO 1 STEP-1 DO

Adjust(A, i, N)

END;

Adjust Algorithm starts from position i . It first checks if the Left child of i index exists. In case yes then it finds if the Right child exists. Out of the Left and the Right child, largest is found. The largest is compared with parent and in case the child has more information, it is exchanged with parent. In case there are more elements below the exchanged one, the process repeats until there is no element below the exchanged child or the exchange with parent is not required.

ALGORITHM Adjust(A[], i, N)

BEGIN:

WHILE $2*i \leq N$ DO

$j \leftarrow 2*i$

IF $j+1 \leq N$ THEN

IF $A[j+1] > A[j]$

$j \leftarrow j+1$

IF $A[j] > A[i]$ THEN

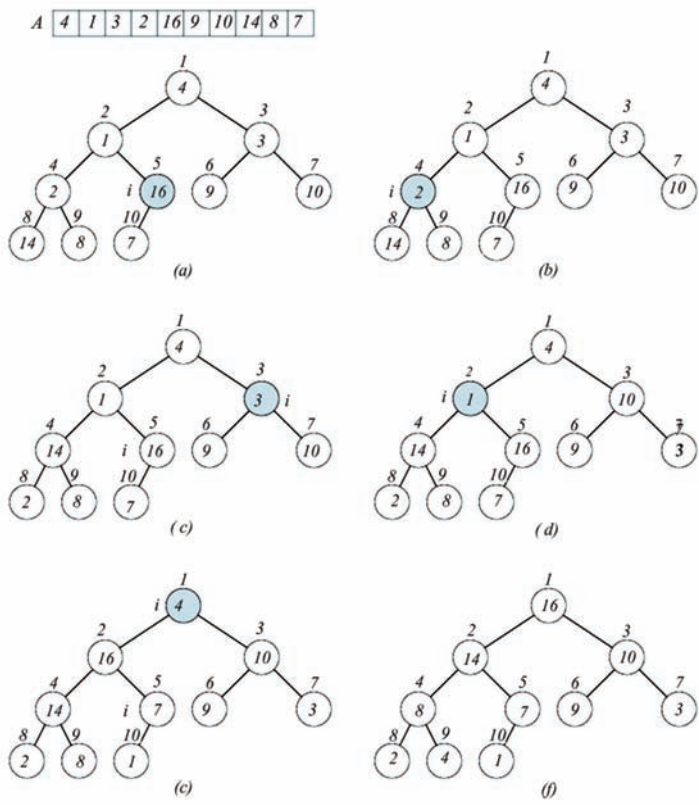
Exchange($A[j]$, $A[i]$)

ELSE

BREAK

$i \leftarrow j$

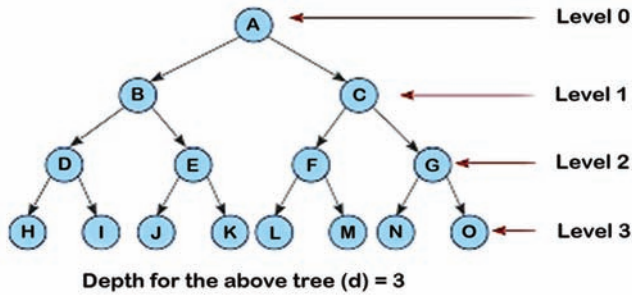
END;



Adjust makes a maximum of $\log_2 n$ exchanges hence the Time complexity of the Adjust is $\Theta(\log_2 n)$

To understand the time complexity of MaxHeapify operation, the concept of level and height is required to be understood.

Level: In a binary tree root has the level 0, its immediate children are at level 1. Children of level 1 are at level 2 and so on so forth. Maximum level of a tree is known as depth



Height: of a node in a binary tree is the distance of a node from Leaf node.

Height 0 Nodes: In the above tree, Nodes H, I, J, K, L, M, N, O are at Height 0

Height 1 Nodes: In the above tree, Nodes D, E, F, G are at Height 1

Height 2 Nodes: In the above tree, Nodes B, C are at Height 2

Height 3 Nodes: In the above tree, Nodes A are at Height 3

Height of the Root node is the height of Tree which is approximately equal to $\log_2 N$ for a Complete Binary Tree.

If a tree has n nodes then the number of nodes height h is given by the formula:

Ceil ($n/2^{h+1}$)

In the above tree $n=15$

For height 0 nodes= Ceil ($15/2$) = 8

For height 1 nodes= Ceil ($15/2^2$) = 4

For height 2 nodes= Ceil ($15/2^3$) = 2

For height 3 nodes= Ceil ($15/2^4$) = 1

Total effort by Max Heapify operation:

= Sum(number of nodes at some height * Effort for Adjust)

= $n/2^1 * C + n/2^2 * 2 * C + n/2^3 * 3 * C + \dots + n/2^{\log n} * \log n * C$

(at height 0, no adjust,

at height 1, $1 * C$ effort for adjust

At height 2, $2 * C$ effort for adjust

...

At height $\log_2 n$, $\log_2 n * C$ effort)

= $n * C (1/2^1 + 2/2^2 + 3/2^3 + \dots + \log n / 2^{\log n})$

<= $n * C (1/2^1 + 2/2^2 + 3/2^3 + \dots \text{infinite terms})$

= $n * C * 2$

Complexity can be written as $\Theta(n)$

Time Complexity: $\Theta(n)$

Space Complexity: $\Theta(1)$

In MaxHeapify, only one variable i is used. In Adjust another j , and temporary variables are required. Total Space is Constant.

To create the Min Heap, MinHeapify operations are performed just as the MaxHeapify was performed.

ALGORITHM MinHeapify(A[], N)

BEGIN:

FOR $i \leftarrow N/2$ TO 1 STEP-1 DO

Adjust(A, i , N)

END;

ALGORITHM Adjust(A[], i , N)

BEGIN:

WHILE $2*i \leq N$ DO

$j \leftarrow 2*i$

IF $j+1 \leq N$ THEN

IF $A[j+1] < A[j]$

$j \leftarrow j+1$

IF $A[j] < A[i]$ THEN

Exchange($A[j]$, $A[i]$)

ELSE

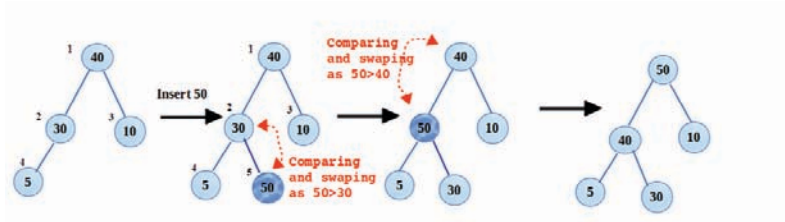
BREAK

$i \leftarrow j$

END;

7.2. INSERTION IN HEAP

If the last element in the Heap is at index N , new element is added at $N+1$ Index. The Inserted element is compared with its parent to reach to the parent in case element is larger. The process is repeated for exchanged parent and parent of parent until either no exchange is required or root has been attained.



ALGORITHM InsertHeap(A[], N, key)

BEGIN:

$A[N+1] \leftarrow \text{key}$

$i \leftarrow N+1$

WHILE $i > 1$ **AND** $A[i] > A[i/2]$ **DO**

$\text{Exchange}(A[i], A[i/2])$

$i \leftarrow i/2$

$N \leftarrow N+1$

END;

Time Complexity: $\Theta(\log_2 N)$

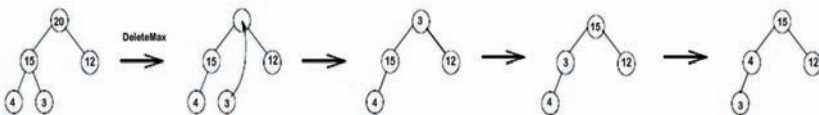
We Traverse up to the heap top to fix the heap property on Insertion using the while loop and it keeps on breaking with $i = i/2$

Space Complexity: $\Theta(1)$

Only i is taken as an extra variable

7.3. DELETION IN HEAP

For Deletion, root node is Deleted and node at index N is placed at index 1, i.e., root. Now it is considered that there are only $N-1$ Nodes. Because of placement of new data value at root, heap property is disturbed at subtree starting at 1. All subtrees are still the Heap. Hence Adjust from 1 is called to make it heap again.



ALGORITHM DeleteHeap(A[], N)

BEGIN:

$\text{key} \leftarrow A[1]$

```

A[1]  $\leftarrow$  A[N]
    Adjust(A, 1, N-1)
N  $\leftarrow$  N - 1
END;

```

Time Complexity: $\Theta(\log_2 N)$

When the element is removed from the top Adjust is called whose complexity is $\log_2 N$. Hence it is $3 + \log_2 N$

Space Complexity: $\Theta(1)$

The only new variable is key. Adjust may require some extra variable. Overall requirement of variables is constant.

7.4. HEAP AS PRIORITY QUEUE

Heaps can be realized as Priority Queue. It is considered that key in the nodes in the heap represents Priority.

If repeated Deletions in Max Heap is performed, it gives the numbers in Descending sequence. Hence it can be renamed as Descending Priority Queue

If repeated Deletions in Min Heap is performed, it gives the numbers in Ascending sequence. Hence it can be renamed as Ascending Priority Queue

Insertions and deletions in the ascending priority queue (min heap are given as):

ALGORITHM PQInsert(A[], N, key)

BEGIN:

```

    A[N+1]  $\leftarrow$  key
    i  $\leftarrow$  N+1
    WHILE i>1 AND A[i]<A[i/2] DO
        Exchange(A[i], A[i/2])
    i  $\leftarrow$  i/2
    N  $\leftarrow$  N+1

```

END;

Time Complexity: $\Theta(\log_2 N)$

Space Complexity: $\Theta(1)$

ALGORITHM ExtractMin(A[], N)**BEGIN:**key $\leftarrow$ A[1]A[1] $\leftarrow$ A[N]

Adjust(A, 1, N-1)

N $\leftarrow$ N - 1**END;****Time Complexity:** $\Theta(\log_2 N)$

When the element is removed from the top Adjust is called whose complexity is $\log_2 N$. Hence it is $3 + \log_2 N$

Space Complexity: $\Theta(1)$

The only new variable is key

7.5. HEAP SORT

For performing Heap Sort, first a Max heap is built. After this Maximum element achieved at root is moved to last index Last index element at root. Since largest element is at last index in the sorted array, forget this element and perform the Adjust from index 1 to make the subtree starting at root to become heap again. The same process is repeated until we remain with 1 element only in the Heap.

ALGORITHM HeapSor(A[], N)**BEGIN:**

MaxHeapify(A, N)

FOR j $\leftarrow$ N TO 2 STEP-1 DO

Exchange(A[1], A[j])

Adjust(A, 1, j-1)

END;

Adjust takes $O(\log_2 N)$ statements, Exchange 3. Loop iterates $N-1$ times. Total statements in inside Loop $(C \log_2 N + 3) * (N-1)$

For Heap Sort total statements = MaxHeapify+repeated exchange and adjust in the loop.

$$= C * N + (C \log_2 N + 3) * (N-1)$$

$$= C * N + CN \log_2 N + 3N - 3 - C \log_2 N$$

Dominating factor here is $N \log_2 N$ hence complexity will be written w.r.t. $N \log_2 N$.

Time Complexity: $\Theta(N \log_2 N)$

Space Complexity: $\Theta(1)$

C-Program

```

/*****/
#include<stdio.h>
/*****/
void HeapSort(int[ ], int);
void Adjust(int[ ], int, int);
void MaxHeapify(int[ ], int);
/*****/
main()
{
    int n, i, a[10];
    printf("enter the size of array : ");
    scanf("%d", &n);
    printf("enter the elements of an array :\n");
    for(i=1; i <= n; i++)
    {
        scanf("%d", &a[i]);
    }
    HeapSort(a, n-1);
    printf("sorted array is\n");
    for(i=1; i <= n; i++)
    {
        printf("%d\n", a[i]);
    }
}

/*****/
void HeapSort(int a[ ], int n)
{

```

```
int j, t, k=10;
MaxHeapify(a, n);
for(j=n; j >= 2; j--)
{
    t=a[j];
    a[j]=a[j/2];
    a[j/2]=t;
    Adjust(a, 1, j/2);
}
}
/*****/
void Adjust(int a[], int i, int n)
{
    int j, t;
    while(2*i <= n)
    {
        j=2*i;
        if (j+1 <= n)
        {
            if(a[j+1]>a[j])
                j=j+1;
        }
        if(a[j]>a[i])
        {
            t=a[j];
            a[j]=a[i];
            a[i]=t;
        }
        else
            break;
        i=j;
    }
}
```



```
/******  
void MaxHeapify(int a[ ], int n)  
{  
    int i;  
    for(i=n/2;i >= 1;i--)  
    {  
        Adjust(a, i, n);  
    }  
}  
/******Output*****/  
enter the size of array : 5  
enter the elements of an array :  
76  
45  
23  
112  
21  
sorted array is  
21  
23  
45  
76  
112
```

EXERCISES

1. If the index of array starts from 0, answer the following
 - a. If child is at i index, what will be the index of parent?
 - b. If parent is at i index, what will be the index of Left and Right Child?
2. Can Heap be implemented using Linked List? Justify your answer.
3. Create a Max Heap for the following data items and perform the Heap Sort 50, 10, 20, 90, 70, 60, 40, 30, 100, 80
4. Consider an Ordered Heap in which each left element is less than its right sibling. Write an Algorithm to create such heap from the given data elements.
5. The array set <A V L T R E I S O K> after the Heapify heap operation changes to

-
6. Match the elements of Table 1 with 2

Table 1

Table 2

- | | |
|-------------------------------|---------------------|
| a) Bubble Sort (Best case) | i) $O(n^2)$ |
| b) Heap sort | ii) $O(n \log_2 n)$ |
| c) Quick sort (Worst case) | iii) $O(1)$ |
| d) Binary Search (worst case) | iv) $O(n)$ |

7. Arrange the following algorithm in increasing order in terms of worst case run time
 - a. Sequential search
 - b. Bubble sort
 - c. Binary search
 - d. Heap sort
8. What is the relation of left and right son of a node in the binary max heap?
 - a. Left son has less information that right son
 - b. Left son has more information that right son
 - c. Left son has equal information to the right son
 - d. There is no relation between left and the right son
9. What is the relation of left and right son of a node in the binary min heap?
 - a. Left son has less information that right son
 - b. Left son has more information that right son
 - c. Left son has equal information to the right son
 - d. There is no relation between left and the right son

10. **What is the property of binary max heap?**
 - a. A node has higher information than its descendants
 - b. A node has less information than its descendants
 - c. A node has equal information to its descendants
 - d. None of these
11. **What is the property of binary min heap?**
 - a. A node has higher information than its descendants
 - b. A node has less information than its descendants
 - c. A node has equal information to its descendants
 - d. None of these
12. **The shape of a binary heap is**
 - a. Strictly binary tree
 - b. Complete binary tree
 - c. Almost complete binary tree
 - d. None of these
13. **The deletion of a node with maximum key in a binary max heap requires**
 - a. $O(n)$ time
 - b. $O(\log n)$ time
 - c. $O(1)$ time
 - d. $O(n \log n)$ time
14. **Adjust (A[], i, n) operation is used to**
 - a. Adjust the shape of subtree rooted at i
 - b. Adjust the height of the tree
 - c. Adjust the heap order property in the subtree rooted at i
 - d. None of these
15. **Let the depth of a heap be 5. What is the minimum no of keys in heap?**
 - a. 33
 - b. 32
 - c. 64
 - d. 63
16. **Let the depth of a heap be 5. What is the minimum no of keys in heap?**
 - a. 33
 - b. 32

-
- c. 64
 - d. 63
17. A Max-heap is constructed by the elements of an array [10, 90, 60, 30, 40, 100, 110, 70]. After the heapify operation, the array changes to.
- a. [110, 90, 100, 70, 40, 10, 60, 30]
 - b. [110, 100, 90, 70, 60, 40, 30, 10]
 - c. [10, 30, 40, 60, 70, 90, 100, 110]
 - d. None of these
18. In a binary heap, the decrease-key (p, x) operation decreases the key of the node pointed by pointer p by x if $x < \text{key}(p)$. What is the maximum no of comparisons required for this operation to run successfully?
- a. $O(n)$
 - b. $O(\log n)$
 - c. $O(1)$
 - d. $O(n \log n)$

8

Matrix Operations

CONTENTS

8.1. Matrix Transposition.....	114
8.2. Printing Matrix In Spiral Form	117
8.3. Matrix Addition	118
8.4. Matrix Multiplication.....	120
Exercise	122

8.1. MATRIX TRANSPOSITION

Matrix transposition is done by exchanging the $A[i][j]$ element with $A[j][i]$. Result is saved in a new matrix.

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}^T$$

ALGORITHM: MatrixTranspose(A[][], M, N)

BEGIN:

B[N][M]

FOR i ← 1 TO M DO

 FOR j ← 1 TO N DO

 B[j][i] ← A[i][j]

RETURN B

END;

Time Complexity: $\Theta(M*N)$

Two nested loops clearly execute a statement $M*N$ times

Space Complexity: $\Theta(M*N)$

A new matrix of size $M*N$ is formed plus two variables i and j are used

C-Program

```

/*****/
#include <stdio.h>
/*****/

int B[10][10];

void MatrixTranspose(int A[ ][3], int M, int N)
{
    for (int i=0; i<M; i++)
    {
        for (int j=0; j<N; j++)
        {
            B[j][i]=A[i][j];
        }
    }
}

```

```

    }
}
/*****/

int main(int argc, constchar * argv[ ]) {
int A[3][3]={ {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };
int m=3;int n=3;
MatrixTranspose(A, m, n);
printf("matrix A→\n\n");
for (int i=0; i<m; i++)
{
for (int j=0; j<n; j++)
{
printf("%d\t", A[i][j]);
}
printf("\n");
}
printf("\n\ntranspose of matrix A→\n\n");
for (int i=0; i<m; i++)
{
for (int j=0; j<n; j++) {
printf("%d\t", B[i][j]);
}
printf("\n");
}
return 0;
}
/*****/

```

8.1.1. Matrix Transposition without Using the Second Matrix

Matrix transposition is done by exchanging the $A[i][j]$ element with $A[j][i]$. Result is saved in the same matrix. Elements up to one diagonal are required to be stored.

ALGORITHM: MatrixTranspose(A[][], M, N)

BEGIN:

```

    FOR i ← 1 TO M DO
        FOR j ← 1 TO i DO
            temporary ← A[i][j]
            A[i][j] ← A[j][i]
            A[j][i] ← temporary

```

```

RETURN A

```

END;**Time Complexity:** $\Theta(N^2)$

Three statements are executed $n*n$ times by two nested loops

Space Complexity: $\Theta(1)$

Only two variables i, j are used separately.

C-Program

```

/*****/
#include <stdio.h>
/*****/

void MatrixTranspose(int A[ ][3], int M, int N)
{
    int temporary;
    for (int i=0; i<M; i++) {
        for (int j=0; j<i; j++) {
            temporary=A[i][j];
            A[i][j]=A[j][i];
            A[j][i]=temporary;
        }
    }
}

/*****/

int main(int argc, constchar * argv[ ]) {
    int A[3][3]={ {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };

```

```

int m=3;int n=3;

printf("matrix A→\n\n");
for (int i=0; i<m; i++) {
for (int j=0; j<n; j++) {
printf("%d\t", A[i][j]);
    }
printf("\n");
}
MatrixTranspose(A, m, n);
printf("\n\ntranspose of matrix A→\n\n");
for (int i=0; i<m; i++) {
for (int j=0; j<n; j++) {
printf("%d\t", A[i][j]);
    }
printf("\n");
}
return 0;
}
/*****/

```

8.2. PRINTING MATRIX IN SPIRAL FORM

ALGORITHM MatrixInSpiralForm(a[][], r, c)

BEGIN:

 i, k ← 0, l ← 0

 WHILE k < r and l < c DO

 FOR i ← l TO c DO

 WRITE(a[k][i])

 k++

FOR i ← k TO r DO

 WRITE(a[i][c-1])

 c--

```

IF k<r THEN
FOR i ← c-1 TO 1 DO
    WRITE(a[r-1][i])
r--
IF l<c THEN
FOR i ← r-1 TO k STEP -1 DO
    WRITE("%d ", a[i][l])
l++
END;

```

Time Complexity: $\Theta(r*c)$

Since all the $r*c$ elements are Traversed once, the complexity is $r*c$

Space Complexity: $\Theta(1)$

Only three variables are used extra in any case

8.3. MATRIX ADDITION

Element i, j of one matrix is added with the element i, j of second matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} \end{bmatrix}$$

ALGORITHM: MatrixAdd(A[][], B[][], M, N)

BEGIN:

```

    C[M][N]
FOR i ← 1 TO M DO
    FOR j ← 1 TO N DO
        C[i][j] ← A[i][j]+B[i][j]
RETURN C
END;

```

Time Complexity: $\Theta(M*N)$

Two nested loops clearly execute a statement $m*n$ times

Space Complexity: $\Theta(M*N)$

A new array (matrix C) is formed containing $m*n$ elements

C-Program

```

/*****/
#include <stdio.h>
/*****/

int C[10][10];
void MatrixAdd(int A[ ][3], int B[ ][3], int M, int N)
{
    for (int i=0; i<M; i++) {
        for (int j=0; j<N; j++) {
            C[i][j]=A[i][j]+B[i][j];
        }
    }
}

/*****/
int main(int argc, constchar * argv[ ])
{
    int A[3][3]={ {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };
    int B[3][3]={ {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };
    int m=3;int n=3;
    MatrixAdd(A, B, m, n);
    for (int i=0; i<m; i++) {
        for (int j=0; j<n; j++) {
            printf("%d\t", C[i][j]);
        }
        printf("\n");
    }
    return 0;
}

/*****/

```

8.4. MATRIX MULTIPLICATION

First work across the 1st row of the first matrix, multiplying down the 1st column of the second matrix, element by element. resulting products are added. The answer goes in position *c11* (top Left) of the answer matrix.

$$\begin{array}{c}
 c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \\
 \begin{array}{c}
 \left[\begin{array}{cccc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{array} \right]_{2 \times 4} \left[\begin{array}{ccc} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{array} \right]_{4 \times 3} = \left[\begin{array}{ccc} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{array} \right]_{2 \times 3}
 \end{array}
 \end{array}$$

$$\begin{array}{c}
 c_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41} \\
 \begin{array}{c}
 \left[\begin{array}{cccc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{array} \right]_{2 \times 4} \left[\begin{array}{ccc} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{array} \right]_{4 \times 3} = \left[\begin{array}{ccc} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{array} \right]_{2 \times 3}
 \end{array}
 \end{array}$$

ALGORITHM: MatrixMultiply(A[][], M, N, B[][], P, Q)

// M & N are the rows and Column of First Matrix, P & Q are rows and columns of second matrix

BEGIN:

C[M][Q]

IF N! $\leftarrow$ P THEN

FOR i $\leftarrow$ 1 TO M DO

FOR j $\leftarrow$ 1 TO Q DO

C[i][j] $\leftarrow$ 0

FOR k $\leftarrow$ 1 TO N DO

C[i][j] $\leftarrow$ C[i][j] + A[i][k] * B[k][j]

RETURN C

END;

Time Complexity: $\Theta(N^3)$ {if Matrix is square of size $N \times N$ }

Three nested loops are executing a single statement n times each and initialize an element to zero. Hence, number of statements executed are $n \times n \times n + n \times n + 1$

Space Complexity: $\Theta(N^2)$ {if Matrix is square in size $N \times N$ }

A new array (matrix C) is formed containing $m \times n$ elements

C-Program

```
/******  
#include <stdio.h>  
/******  
int C[10][10];  
void Matrixmultiply(int A[ ][3], int M, int N, int B[ ][3], int P, int Q)  
{  
    if (N == P) {  
        for (int i=0; i<M; i++) {  
            for (int j=0; j<Q; j++) {  
                C[i][j]=0;  
                for (int k=0; k<N; k++) {  
                    C[i][j]=C[i][j]+A[i][k]*B[k][j];  
                }  
            }  
        }  
    }  
}  
/******  
int main(int argc, constchar * argv[ ]) {  
    int A[3][3]={ {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };  
    int B[3][3]={ {1, 2, 3}, {1, 2, 3}, {1, 2, 3} };  
    int m=3;int n=3;int p=3, q=3;  
    Matrixmultiply(A, m, n, B, p, q);  
    for (int i=0; i<m; i++) {  
        for (int j=0; j<n; j++) {  
            printf(“%d\t”, C[i][j]);  
        }  
        printf(“\n”);  
    }  
    return 0;  
}  
/******
```

EXERCISE

Programming Questions

1. Write a Program for finding determinant of a Matrix
2. Write a Program to find the inverse of the given matrix
3. Write a program to Find the largest among the Matrix Element
4. Write a Program to sort the Matrix Content
5. Write a Program to find the Matrix product in $O(N^2)$ time.

9

Strings

CONTENTS

9.1. Application Problems Related To Strings	124
Exercises.....	137

Definition: Strings are a special type of character array in which last element is null character. Strings are very important Data structure which is utilized in various computer science applications.

char Str[]="Hello";

Index	0	1	2	3	4	5
Value	H	e	l	l	o	'\0'
Address	1000	1001	1002	1003	1004	1005

Above is the string the C programming Language.

Various operations on string is depicted in this chapter.

9.1. APPLICATION PROBLEMS RELATED TO STRINGS

1. Finding Length of String

ALGORITHM LengthOfString(String str[])

BEGIN:

$i \leftarrow 0$

WHILE str[i] != '\0' DO

$i \leftarrow i + 1$

RETURN i

END;

Time Complexity: $\Theta(L)$

It takes L statements to traverse the whole string plus the two statements, one above loop and one below it. L is the length of string.

Space Complexity: $\Theta(1)$

Only i is taken as an extra variable

C-Program

```
/**/
```

```
#include<stdio.h>
```

```
/**/
```

```
int StringLength(char[ ]);
```

```
/**/
```

```
int main()
```

```

{
    char s[100];
    int d;
    printf("enter the string: ");
    scanf("%s", s);
    d=StringLength(s);
    printf("The length of string is %d", d);
    return 0;
}
/*****/
int StringLength(char s[ ])
{
    int i, l=0;
    for(i=0;s[i]!='\0';i++)
    {
        l++;
    }
    return l;
}
/*****/

```

2. Reversing the Given String

ALGORITHM ReverseOfString(STRING str1[])

BEGIN:

```

    STRING str2[ ]
    L ← LengthOfString( str 1[ ] )
    i ← 0
    WHILE i< L DO
        str2[ i ] ← str1[ L ]
        i ← i + 1
    L ← L - 1
    str2[i] ← '\0'

```

```
RETURN str2
```

```
END;
```

Time Complexity: $\Theta(L)$

The statements executed are $3*L$ to reverse the whole string in the loop plus five statements, 3 above the loop and two below it

Space Complexity: $\Theta(L)$

A new string of the same length of string L is used plus two extra variables

C-Program

```

/*****/
#include<stdio.h>
/*****/
int StringLength(char[ ]);
void Reverse(char[ ], int);
/*****/
int main()
{
    char s[100];
    int d;
    printf("enter the string: ");
    scanf("%s", s);
    d=StringLength(s);
    printf("The length of string is %d", d);
    Reverse(s, d);
    return 0;
}
/*****/
int StringLength(char s[ ])
{
    int i, l=0;
    for(i=0;s[i]!='\0';i++)
    {
        l++;
    }
    return l;
}
/*****/
void Reverse(char s[ ], int l)
{
    int i, j;

```

```

        char c[100];
for(j=0, i=l-1; j<l; ++j, --i)
    {
        c[j]=s[i];
    }
    c[l]='\0';
    printf("\nReverse string is %s", c);
}
/*****/

```

3. To Check if the Given String is a Palindrome

Algorithm StringPalindrome(String str[])

BEGIN:

```

    L ← LengthOfString( str[ ] )
    i ← 0
    c ← 0
    WHILE i < L/2 DO
        IF str[ i ] != str[ L-i ]
            c ← c+1
        i ← i+1

```

BREAK

```

    IF c == 0 THEN
        RETURN TRUE
    ELSE
        RETURN FALSE

```

END;

Time Complexity: $\Theta(L)$

The loop runs for $L/2$ times and executed 2 statements in each iteration. Three statements above the loop and two statements conditionally below it. Total $2*L/2+3+2 = L+5$

Space Complexity: $\Theta(1)$

New variable c, L, i are used

C-Program

```
/******/  
#include<stdio.h>  
/******/  
int StringLength(char[ ]);  
void Palindrome(char [ ], int);  
/******/  
int main()  
{  
    char s[100];  
    int d;  
    printf("enter the string: ");  
    scanf("%s", s);  
    d=StringLength(s);  
    printf("The length of string is %d", d);  
    return 0;  
}  
/******/  
int StringLength(char s[ ])  
{  
    int i, l=0;  
    for(i=0;s[i]!='\0';i++)  
    {  
        l++;  
    }  
    return l;  
}  
/******/  
void Palindrome(char s[ ], int l)  
{  
    int i;  
    int c=0;  
    for(i=0;i<l/2;i++)  
    {
```

```

        if(s[i]!=s[l-i-1])
        {
            c++;
            break;
        }
    }
    if(c == 0)
        printf("\nString is palindrome");
    else
        printf("\nString is not palindrome");
}
/*****/

```

4. To Find Number of Words in a Paragraph

Algorithm Word Count(String str[])

BEGIN:

```

    c ← 1
    i ← 0
    WHILE str[ i ] != '\0' DO
IF str[ i ] == ' ' //White space
    c ← c + 1
    i++

```

RETURN c + 1

END;

Time Complexity: $\Theta(N)$, **N is the length of string**

Each element is Traversed to count the number of spaces where we have N elements

Space Complexity: $\Theta(1)$

Two new variables i and c are taken extra

C-Program

```

/*****/
#include <stdio.h>

```

```

#define MAX_SIZE 100 // Maximum string size
/*****
int main()
{
    char str[MAX_SIZE];
    int i, words;

    printf("Enter any string: ");
    gets(str);

    i = 0;
    words = 1;
    while(str[i] != '\0')
    {
        if(str[i] == ' ' || str[i] == '\n' || str[i] == '\t')
        {
            words++;
        }
        i++;
    }
    printf("Total number of words = %d", words);

    return 0;
}
*****/

```

5. To Convert Lower Case letters to Upper Case and vice versa

Algorithm CaseChange(String str[])

BEGIN:

WHILE str[i] != '\0' DO

IF s[i] <= 'z' && s[i] >= 'a' THEN

s[i] ← s[i] – 32

ELSE IF s[i] <= 'Z' && s[i] >= 'A' THEN

```
s[i] ← s[i] + 32
```

END;

Here it is assumed that encoding is the ASCII. In case of Unicode encoding, appropriate changes would be applicable

C-Program

```
/******  
#include<stdio.h>  
/******  
void exchanger(char s[ ])  
{  
    int i;  
    char l[5000], u[5000];  
    for(i=0;s[i]!='\0';i++)  
    {  
        if(s[i] <= 'z' && s[i] >= 'a')  
            s[i]-=32;  
        else if(s[i] <= 'Z' && s[i] >= 'A')  
            s[i]+=32;  
    }  
    puts(s);  
}  
/******  
void main()  
{  
    char s[5000];  
    printf("Enter the string:\n");  
    gets(s);  
    exchanger(s);  
}  
/****** Output *****/
```

Enter the string:

QWERTY keyboard

qwerty KEYBOARD

/******

6. To Arrange the Strings in Dictionary Order

ALGORITHM Dictionary order(str[][], Count) // count is the number of strings

BEGIN:

FOR i=0 TO Count DO

 FOR j=i+1 to Count DO

 IF StringCompare(str[i], str[j])>0 THEN

 Exchange(str[i], str[j])

END;

C-Program

/******

#include<stdio.h>

#include<string.h>

/******

int main(){

 int i, j, count;

 char str[25][25], temporary[25];

 printf("How many strings u are going to enter?: ");

 scanf("%d", &count);

 printf("Enter Strings one by one: ");

 for(i=0;i <= count;i++)

 gets(str[i]);

 for(i=0;i <= count;i++)

 {

 for(j=i+1;j <= count;j++)

 {

 if(strcmp(str[i], str[j])>0)

 {

```

        strcpy(temporary, str[i]);
        strcpy(str[i], str[j]);
        strcpy(str[j], temporary);
    }
}

printf("Order of Sorted Strings:");
for(i=0;i <= count;i++)
puts(str[i]);
return 0;
}

```

/******OUTPUT******/

How many strings u are going to enter?: 5

Enter Strings one by one: John

Rose

Twinkle

Herry

Suzzie

Order of Sorted Strings:

Herry

John

Rose

Suzzie

Twinkle

/*******/

7. To Reverse all Words in a Message

ALGORITHM ReverseWords(msg[])

BEGIN:

i ← 0

j ← 0

```

WHILE msg[i] != '\0' DO
    IF msg[i] != ' ' THEN
        str[j] ← msg[i]
        j++
    ELSE
        str[j] ← '\0'
WRITE(str)
    j ← 0
    i++
END;

```

C-Program

```

/*****/
#include<stdio.h>
/*****/

void main() {
    char msg[ ] = "Welcome to Programming World";
    char str[1000];

    int i = 0; int j = 0;

    while (msg[i] != '\0')
    {
        if (msg[i] != ' ')
        {
            str[j] = msg[i];
            j++;
        }
        else
        {
            str[j] = '\0';
            printf("%s", str);
            printf(" ");
            j = 0;
        }
    }
}

```

```

        }
        i++;
    }
}

```

/********OUTPUT*******/

emocleW ot gnimmargorP dlroW

8. To Search the Location of the Word in the Given Text

Algorithm FindLocation(s1[], s2[])

BEGIN:

M ← StringLength(s1)

N ← StringLength(s2)

FOR i ← 0 to i ← N-M DO

 FOR j ← 0 to M DO

 IF s2[i + j] != s1[j] THEN

 J ← M+1

 IF j == M THEN

 RETURN i

END;

C-Program

```
#include <stdio.h>
```

```
/******
```

```
int StringLength(char s[ ])
```

```
{
```

```
    int i, l=0;
```

```
    for(i=0;s[i]!='\0';i++)
```

```
    {
```

```
        l++;
```

```
    }
```

```
    return l;
```

```
}
```

```

/*****

```

```

int isSubstring(char s1[ ], char s2[ ])

```

```

{

```

```

    int M = StringLength(s1);

```

```

    int N = StringLength(s2);

```

```

    for (int i = 0; i <= N - M; i++)

```

```

    {

```

```

        int j;

```

```

        for (j = 0; j < M; j++)

```

```

            if (s2[i + j] != s1[j])

```

```

                j=M+1

```

```

        if (j == M)

```

```

            return i;

```

```

        }

```

```

        return -1;

```

```

    }

```

```

/*****

```

```

int main()

```

```

{

```

```

    char s1[ ] = "abes";

```

```

    char s2[ ] = "abes engineering college";

```

```

    int res = isSubstring(s1, s2);

```

```

    if (res == -1)

```

```

        printf("Not present");

```

```

    else

```

```

        printf("Present at index %d", res);

```

```

    return 0;

```

```

}

```

```

/*****OUTPUT*****/

```

```

Present at index 0

```

```

/*****

```

EXERCISES

Programming Questions

1. Write Program in C for searching a keyword in a text. The keyword may appear multiple times.
2. Write Program in C for searching a record related to name in the table
3. Write a Program in C for counting the frequency of various characters in a text message.

10

Stack

CONTENTS

10.1. Stack Basics	140
10.2. Primitive Operations On Stack.....	141
10.3. Applications Of Stack	145
Exercises.....	183

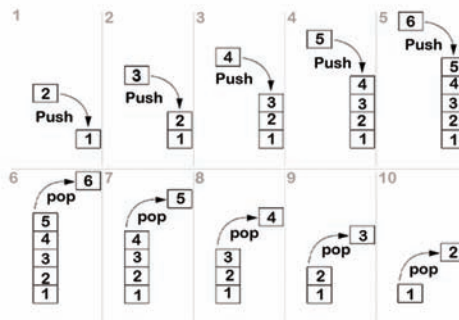
10.1. STACK BASICS

Stacks are ordered a collection of items into which items may be inserted or removed from the same END called the top of the stack. Stack works on **Last in First Out** principle means an item which is inserted first will be taken out at last and vice versa.

10.1.1. How to Understand a Stack Practically?

There are many real-life examples of the stack. Consider the simple example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e., the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

There are some basic terminology used to describe the Primitive operations of stack such as **Push** means to increase the size of stack by Inserting an element into it and **Pop** means to remove an element from the stack.(note: both the Push and Pop operations are possible only at the topmost element of the stack.). Other primitive operations such as searching, sorting, and traversal are not possible in the stack as we can only access a topmost element of the stack.



10.1.2. Applications of Stack

- Checking validity of expression;
- Arithmetic and Logic expression evaluations and inter-conversions;
- Forward and backward feature in web browsers;
- Syntax parsing;
- Back tracing;
- Resolving function calls;
- Resolving recursion;
- Managing undo operations in the Software;
- Visibility of received WhatsApp messages/emails etc.;

- Reversing the String/number;
- Palindrome check;
- Conversion of different base numbers.

10.2. PRIMITIVE OPERATIONS ON STACK

In this section, it is assumed that stacks are implemented using Array. There are two components of the Stack. One the Array in which elements will be Inserted and TOP that denotes the topmost index of the array at the moment.

STACK is a type and if its variable is taken S

STACK S

Top can be accessed at S.TOP

And Stack Element can be accessed as S.ITEM [S.TOP]

- 1. Initialize Stack:** For array implementation of Stack, TOP is treated as the Index of array. Index 0 is the valid index and SIZE-1 is the last index in the array. If Top is initialized at 0, it will depict that there is one element in the Stack hence it is initialized to invalid index, i.e., -1.

ALGORITHM InitializeStack(Stack S)

BEGIN:

S.TOP=-1

END;

Time Complexity: $\Theta(1)$

There is only one instruction in the Algorithm

Space Complexity: $\Theta(1)$

There are no extra variable taken in the Algorithm

- 2. Insertion in Stack (Push):** In case the stack is already full, further Insertion will not be possible. An attempt to Insert an element in the Full stack will lead to Overflow. This is the erroneous condition and should not have happened. The message/information about its occurrence should be sent to EXIT and program should terminate.

$\Rightarrow$ In a bucket full of Water, if more water is poured in, the inlet water will fall down on the floor. This is the overflow.

If all goes well, simply update the TOP index and Insert the element at TOP index in the array.

ALGORITHM Push(Stack S, key)**BEGIN:**

IF S.TOP == SIZE-1 THEN

WRITE("Stack Overflows")

EXIT (1)

ELSE

S.TOP++

 S.ITEM[S.TOP] $\leftarrow$ key**END;****Time Complexity: $\Theta(1)$**

In case the TOP is equal to SIZE-1, Two statements inside the condition is executed. In case this is false then two statements are executed in else. Three statements are executed conditionally.

Space Complexity: $\Theta(1)$

No extra variable is taken

- 3. Empty Check:** In case there is no element in the stack, the stack will be Empty. The Empty check is the boolean value function that will either return TRUE or FALSE.

ALGORITHM Empty (Stack S)**BEGIN:**

IF S.TOP == -1 THEN

RETURN TRUE

ELSE

RETURN FALSE

END;**Time Complexity: $\Theta(1)$**

Two statements are executed conditionally

Space Complexity: $\Theta(1)$

No extra variable is utilized

- 4. Deletion in Stack (POP):** Deletion has been given standard name 'POP.' In case the stack is Empty, deletions will not be permissible. An attempt to Delete an element from the Empty stack will lead to Underflow. This is the erroneous condition and should not have hap-

pened. The message/information about its occurrence should be sent to EXIT and program should terminate.

If all goes well, simply save the TOP element and update the TOP index by decrementing it by 1.

ALGORITHM POP (Stack S, key)

BEGIN:

IF EMPTY(S) THEN
WRITE ("Stack underflows")
EXIT(1)

ELSE

$X \leftarrow S.ITEM[S.TOP]$
S.TOP- -
RETURN X

END;

Time Complexity: $\theta(1)$

In case the stack is Empty, Two statements inside the condition is executed. In case this is false then three statements are executed in else. 3/4 statements are executed conditionally

Space Complexity: $\theta(1)$

No extra variable is taken

5. **Reading TOP element in Stack (StackTop):** It reads the top element and returns.

ALGORITHM StackTop (Stack S)

BEGIN:

RETURN (S.ITEM[S.TOP])

END;

Time Complexity: $\theta(1)$

Only one statement is executed

Space Complexity: $\theta(1)$

No extra variable is used

C-Program

```
/******
```

```
#include<stdlib.h>
```

```
#define SIZE 10
#define TRUE 1
#define FALSE 0

struct stack
{
    int item[SIZE];
    int top;
}S;
/*****/

void initialize( )
{
    S.top=-1;
}
/*****/

void Push( int key)
{
    if(S.top == SIZE-1)
    {
        printf("stack overflows");
        exit(1);
    }
    S.top++;
    S.item[S.top]=key;
}
/*****/

int Empty( )
{
    if(S.top == -1)
        return TRUE;
    else
        return FALSE;
}
/*****/

int Pop( )
```

```

{
    int x;
    if(Empty())
    {
        printf("stack underflows");
        exit (1);
    }
    x=S.item[S.top];
    S.top--;
    return x;
}
/*****
void main()
{
    Push (100);
    Push (200);
    Push (300);
    Push (400);
    Push (500);
    x=POP();
    printf("Deleted element is:=> %d:\n", x);
    x=POP();
    printf("Deleted element is:=> %d:\n", x);
}
/***** Output *****/
Deleted element is 500
Deleted element is 400
/*****/

```

10.3. APPLICATIONS OF STACK

10.3.1. Decimal to Binary, Octal, Hexadecimal Conversion

1. **Decimal to Binary:** Repeated divisions of a decimal number

with 2 is performed and remainder is saved in the stack. Stack is Popped and printed.

ALGORITHM DecimalToBinary(n)

BEGIN:

```

Stack S
Initialize(s)
WHILE n!=0 DO
    r ← n%2
    Push(S, r)
    n ← n/2
WHILE ! Empty (s) DO
    X ← Pop(s)
    Write(x)

```

END;

Time Complexity: $\Theta(N)$

Any number can be divided by 2 $n/2$ times. Hence the loop makes $n/2$ iterations

Space Complexity: $\Theta(1)$

Three new variables r, X, and a stack S are taken

C-Program

```

/*****/
#include<stdlib.h>
#define SIZE 10
#define TRUE 1
#define FALSE 0
/*****/

struct stack
{
    int item[SIZE];
    int top;
}S;

```

```
/******  
void initialize( )  
{  
    S.top=-1;  
}  
/******  
void Push( int key)  
{  
    if(S.top == SIZE-1)  
    {  
        printf("stack overflows");  
        exit(1);  
    }  
    S.top++;  
    S.item[S.top]=key;  
}  
/******  
int Empty( )  
{  
    if(S.top == -1)  
        return TRUE;  
    else  
        return FALSE;  
}  
/******  
int Pop( )  
{  
    int x;  
    if(Empty())  
    {  
        printf("stack underflows");  
        exit (1);  
    }  
    x=S.item[S.top];
```



```

        S.top--;
        return x;
    }
    /*****/
void DecimalToBinary(int n)
{
    int r, x;
    initialize();
    while(n!=0)
    {
        r=n%2;
        Push(r);
        n=n/2;
    }
    printf("binary no. =");
    while(!Empty())
    {
        x=Pop();
        printf("%d", x);
    }
    printf("\n");
}

/*****/
void main()
{
    int n;
    printf("enter the no : \n");
    scanf("%d", &n);
    DecimalToBinary(n);
}
/*****/

```

2. **Decimal to Octal:** Repeated divisions of a decimal number with 2 is performed and remainder is saved in the stack. Stack is Popped and

printed.

ALGORITHM DecimalToOctal(n)

BEGIN:

STACK S

Initialize(s)

WHILE $n \neq 0$ DO

$r \leftarrow n \% 8$

Push(S, r)

$n \leftarrow n / 8$

WHILE ! Empty (s) DO

$X \leftarrow \text{Pop}(s)$

WRITE(x)

END;

Time Complexity: $\Theta(N)$

Any number can be divided by 8 $n/8$ times. Hence the loop makes $n/8$ iterations

Space Complexity: $\Theta(1)$

Three new variables r, X, and a stack S are taken

/******

Structure Declaration and Primitive operations remains the same as in the previous Program

/******

C-Function

void DecimalToOctal(int n)

{

int r, x;

initialize();

while($n \neq 0$)

{

$r = n \% 8$;

Push(r);

$n = n / 8$;

```

}printf("octal no.=");
while(!Empty())
{
    x=Pop();
    printf("%d", x);
}
printf("\n");
}
/*****/
main()
{
    int n;
    printf("enter the no\n");
    scanf("%d", &n);
    DecimalToOctal(n);
}
/*****/

```

3. Decimal to Hexadecimal: Structure Declaration and Primitive operations remains the same as in the previous Program

```

void DecimalToHexadecimal(int n)
{
    int r, x;
    initialize();
    while(n!=0)
    {
        r=n%16;
        Push(r);
        n=n/16;
    }
    printf("hexadecimal no.=");
    while(!Empty())
    {

```

```

        x=Pop();
        if(x<10)
            printf("%d", x);
        Else
    {
switch (x)
{
case 10: printf("A"); break;
case 11: printf("B"); break;
case 12: printf("C"); break;
case 13: printf("D"); break;
case 14: printf("E"); break;
case 15: printf("F"); break;
}
}
}
/*****/

void main()
{
int n;
printf("enter the no\n");
scanf("%d", &n);
DecimalToHexadecimal(n);
}
/*****/

```

4. Decimal to Any Base Conversion

ALGORITHM DecimaltoAnyBase(n, b)

BEGIN:

Stack S

Initialize(s)

A[16]={‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’, ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’}

```

WHILE n!=0 DO
  r ← n%b
  Push(S, A[r])
  n ← n/b
  WHILE ! Empty (s) DO
    X ← Pop(s)
    Write(x)

```

END;

Time Complexity: $\Theta(N)$

Any number can be divided by b n/b times. Hence the loop makes n/b iterations

Space Complexity: $\Theta(1)$

Four new variables r, X, A[16] and a stack S are taken

C-Funtion

```

/*****/
void DecimalToAnyBase(int n, int base)
{
  int r, x;
  int A[16]={ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
  initialize( );
  while(n!=0)
  {
    r=n%base;
    Push(r);
    n=n/base;
  }
  printf("base %d number equivalent is =", base);
  while(!Empty())
  {

```

```

        x=Pop();
        printf("%d", A[x]);
    }
/*****/

void main()
{
    int n, b;
    printf("enter the decimal no\n");
    scanf("%d", &n);

    printf("enter the base in which conversion is required \n");
    scanf("%d", &b);

    DecimalToAnyBase(n);
}
/*****/

```

10.3.2. Reversing the String

Each character of the string is pushed on the stack. When there are no character Left in the string, Start deleting elements from Stack one by one and print them in sequence. Since characters are retrieved from stack in reverse order, string is also reversed.

Here the array taken with the stack is character array.

ALGORITHMString Reverse(String str[])

BEGIN:

```

    i=0
    STACK s
    Initialize (s)
    WHILE str[i] !=""\0"" DO
        Push (s , str[i])
        i++
    WHILE ! Empty(s) DO
        x ← Pop(s)
        WRITE(x)

```

END;

Time Complexity: $\Theta(N)$

The loop runs to execute statements n times. i.e., 1 iteration to for each element where the length of the string is n

Space Complexity: $\theta(N)$

The maximum size of the stack will be n

C Program

```

/*****/
#include<stdio.h>
#define SIZE 20
/*****/
void initialize();
void Push(char);
int Pop();
void reverse();
/*****/
struct STACK
{
    char item[SIZE];
    int top;
} S;
/*****/
void main()
{
    char w[SIZE];
    printf("Enter the string\n");
    gets(w);
    reverse(w);
}
/*****/
void reverse(char *str)
{
    int i=0;
    initialize();
    while(str[i]!='\0')
    {
        Push(w[i]);
        i++;
    }
    printf("Reverse string=");
    while(!Empty(S))
    {
        char x;
    }
}

```

```

    x=Pop();
    printf("%c", x);
}
}
/*****/
void initialize()
{
    s.top=-1;
}
/*****/
void Push(char key)
{
    if(s.top == SIZE-1)
    {
        printf("Stack overflows");
        exit(1);
    }
    s.top++;
    s.item[s.top]=key;
}
/*****/
int Pop()
{
    char x;
    if(s.top == -1)
    {
        printf("Stack Empty");
        exit(1);
    }
    x=s.item[s.top];
    s.top--;
    return x;
}
/*****/
Enter the string
hello
Reverse string=olleh
/*****/

```

10.3.3. Palindrome Check

Just as in reverse of the string characters of the strings are stored on the stack. If deletions are performed, string characters will be obtained in the reverse order. Start comparing the top element of the stack with the string characters starting from

first character. In case there is a mismatch, this indicates the given string is not a palindrome. If all characters do match, and stack is Empty in the last, given string will be a palindrome.

ALGORITHMPalindrome(String str[])

BEGIN:

$i \leftarrow 0$

Stack s

Initialize (s)

WHILE str[i] != ""\0" DO

 Push (s , str[i])

 i++

$j \leftarrow 0$

WHILE ! Empty(s) DO

 IF Str[j] == S.TOP(S) THEN

 Pop(s)

 ELSE

 BREAK

IF Empty (s) THEN

 WRITE ("Palindrome")

ELSE

 WRITE (" Not Palindrome")

END;

Time Complexity: $\Theta(N)$

The loop runs to execute statements n times. i.e., 1 iteration to for each element where the length of the string is n

Space Complexity: $\Theta(N)$

The maximum size of the stack at any point can be n

C-Program

```
/******
```

```
#include<stdio.h>
```

```
#define SIZE 20
/*****/
void initialize();
void Push(char);
int Pop();
void palindrome();
/*****/
struct STACK
{
    char item[SIZE];
    int top;
}S;
/*****/
void main()
{
    char w[SIZE];
    printf("Enter the string\n");
    gets(w);
    palindrome(w);
}
/*****/
void palindrome(char *str)
{
    int i=0;
    initialize();
    while(str[i]!='\0')
    {
        Push(str[i]);
        i++;
    }
    int j=0;
    while(s.top!=-1)
    {
        if(str[j] == s.item[s.top])
```

```

    {
        Pop();
        j++;
    }
    else
        break;
}
if(Empty(S))
    printf("\nPalindrome\n");
else
    printf("\nnot Palindrome\n");
}
/*****/

Enter the string
madam

Palindrome
/*****/

```

10.3.4. Expression Validity Check on the Basis of Parenthesis Position

Given an arithmetic expression-containing parenthesis. e.g., A+ (B/C). The task is to check if the given expression is correct according to the bracket positions. For this assume the expression-containing the only parenthesis.

- 1.)
- 2.()
- 3.())
- 4.()()
- 5.)(

Above are some of the cases of unbalanced or invalid parenthesis order

Solution:

Start with AN Empty stack

Push upon occurrence of opening parenthesis (

Upon occurrence of closing parenthesis, check if the stack is Empty or not. In case it is Empty, Pop the stack. In case it is not Empty, meaning that the closing parenthesis are more than opening and declare the invalid expression

If there is no other character Left in the expression, Check if the stack is Empty. If the stack is Empty meaning that Expression is valid. The expression started with the opening parenthesis and there were exactly the same number of closing parenthesis as the opening ones.

In case stack is not Empty meaning that number of opening parenthesis were more than closing ones. The expression is invalid.

ALGORITHM ValidExpression(String Exp[])

BEGIN:

Valid=1

Stack S

Initialize S

WHILE Exp[i] != "\0" DO

IF Exp[i] == '(' THEN

Push (S, Exp[i])

ELSE

IF Exp[i] == ')' THEN

IF Empty (S) THEN

Valid ← 0

BREAK

ELSE

Pop(S)

i++

IF valid == 0 THEN

WRITE("Invalid Expression")

ELSE

IF Empty(S) THEN

WRITE("Valid Expression")

ELSE

WRITE("Invalid Expression")

END;

Time Complexity: $\Theta(N)$

The loop runs to execute statements n times. i.e., 1 iteration to for each element

Space Complexity: $\Theta(N)$

The maximum size of the stack at any point can be $n/2$

C-Program

```

/*****/
#include<stdio.h>
#define SIZE 20
/*****/
void initialize();
void Push(char);
int Pop();
void paranthesis();
/*****/
struct
{
    char item[SIZE];
    int top;
}s;
/*****/
main()
{
    char w[SIZE];
    printf("Enter the expression\n");
    gets(w);
    ParenthesisCheck(w);
}
/*****/
void ParenthesisCheck(char *str)
{
    int i=0, valid=1;
    initialize();

```

```
while(str[i]!='\0')
{
    if(str[i] == '(')
        Push(str[i]);
    else
    {
        if(str[i] == ')')
        {
            if(Empty())
            {
                valid=0;
                break;
            }
            else
                Pop();
        }
    }
    i++;
}
if(valid == 0)
    printf("Invalid Expression\n");
else
{
    if(Empty())
        printf("valid Expression\n");
    else
        printf("Invalid Expression\n");
}
```

/******OUTPUT******/

Enter the expression

(a+b)(c*c)(

Invalid Expression

Enter the expression

(a+b)(c*c)

valid Expression

/*****/

10.3.5. Inter-Conversion and Evaluation of Polish Notation Expression

Suppose we have been given an expression:

A+B*C-D/E+F↑G-(H/G)

To solve such expression, we need to Traverse the entire expression to find which of the operator has got the highest precedence. We need to go from Left Right and Right to Left multiple time to evaluate the expression.

In case we are given another type of expression (Postfix), e.g., 897-*, If we solve this like:

- If operands are observed, push
- If operators are observed, pop stack twice and store elements in b and a respectively. Apply the operator on a & b and Push the result in the stack.
- If no term Left in the expression, Pop the stack and this is the result.

In this case, we only need to move from Left to Right once. As soon as we reach the Right end, we have the answer.

					7	Top						
	Top		9	Top	9			2	Top			
8			8		8			8			16	Top
8			9		7			-			*	
								b=9			b=8	
Push(8)			Push(9)		Push(7)			a=7			a=2	
								a-b=2			2*8=16	
								Push(2)			Push(16)	

There are three fashions in which expression can be written:

Type of Expression	Description	Example
--------------------	-------------	---------

Infix	Operator is written in between the Operands	3+4
Postfix	Operator is written after the Operands	34+
Prefix	Operator is written before the Operands	34

The prefix is called polish notation expression, and postfix is called reverse polish notation expression.

If stacks are used for Evaluation of Prefix and Postfix Expression, Expression evaluation becomes very simple and straightforward. Precedence and associativity of operators are not required to be checked while evaluating the expression using stack.

Example: $4325^*+ = 4+3-2*5 = -3$

Symbol	opnd1	opnd 2	value	opndstack
4				4
3				4,3
2				4, 3,2
5				4,3, 2,5
*	2	5	10	4, 3
				4, 3, 10
-	3	10	-7	4
				4, -7
+	4	-7	-3	-3



result

ALGORITHM POSTFIX EVALUATION (Postfix Expression)

BEGIN:

STACK OperandStack

Initialize (OperandStack)

WHILE not end of input from postfix expression Do

Symbol $\leftarrow$ Next character from postfix expression

IF symbol is an operand THEN

Push (OperandStack, Symbol)

ELSE

oprnd 2 $\leftarrow$ Pop (OperandStack)

oprnd 1 $\leftarrow$ Pop(OperandStack)

value $\leftarrow$ Result of applying symbol to oprnd1 and oprnd2

Push(OperandStack, value)

Result $\leftarrow$ Pop(OperandStack)

RETURN Result

END;

Time Complexity: $\Theta(N)$

There are N symbols in the Expression. For each symbol, decision is to be taken for Push or Pop. In the case of operand 1 statement execution and for operator 4. There are $N/2+1$ operands and $N/2-1$ operators.

Hence total statements

$$= \text{Initialization} + (N/2+1) * 1 + (N/2-1) * 3 + \text{Return result}$$

$$= 1 + 2*N - 2 + 2$$

$$= 2*N + 1$$

Space Complexity: $\Theta(N)$

$N/2+1$ size stack is required and three variables for Symbol, value, and result. Total space = $N/2+1+3 = N/2+4$

C-Program

```

/*****/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
/*****/
#define SIZE 20
int PostFixEvaluation(char [ ]);
int Evaluation(int , int , char );
/*****/
struct stack
{
    int item[SIZE];
    int top;
}s;
/*****/
void Initialize()
{

```

```
        s.top=-1;
    }
    /*****/
void Push(int key)
{
    if(s.top == SIZE-1)
    {
        printf("stack overflow");
        exit(1);
    }
    s.top++;
    s.item[s.top]=key;
}
    /*****/
int Pop()
{
    char x;
    if(s.top == -1)
    {
        printf("stack underflow");
        exit(1);
    }
    x=s.item[s.top];
    s.top--;
    return x;
}

    /*****/
int PostFixEvaluation(char postfix[ ])
{
    int i=0, op1, op2, value, x;
    char symbol;
```

```

while(postfix[i]!='\0')
{
symbol=postfix[i];
if(symbol >= '0' && symbol <= '9')
Push(symbol-'0');
else
{
op2=Pop();
op1=Pop();
value=Evaluation(op1, op2, symbol);
Push(value);
}
i++;
}
x=Pop();
return x;
}
/*****/
int Evaluation(int op1, int op2, char symbol)
{
int x;
switch(symbol)
{
case '+':      x=(op1+op2);   break;
case '-':      x=(op1-op2);   break;
case '*':      x=(op1*op2);   break;
case '/':      x=(op1/op2);   break;
case '%':      x=(op1%op2);   break;
case '^':      x= pow(op1, op2); break;
}
return x;
}
/*****/
main()

```

```

{
    Initialize();
    char postfix[20]="454+-";
    int answer;
    answer=PostFixEvaluation(postfix);
    printf("%d", answer);
}
/*****/

```

10.3.5.1. Evaluation of Prefix Expression

To evaluate the prefix expression, the following steps are performed:

- Take an empty operator stack;
- Reverse the prefix expression;
- If operands are observed, push the operand on the stack;
- If operators are observed, pop stack twice and store elements in a and b respectively;
- Apply the operator on a & b and push the result in stack.

If no term Left in the expression, pop the stack and this is the result

			9	Top				8	Top				
7	Top		7		2	Top		2			16	Top	
7			9		-			8			*		
Push(7)			Push(9)		a=9			Push(8)			a=8		
					b=7						b=2		
					a-b=2						8*2=16		
					Push(2)						Push(16)		

ALGORITHM PrefixEvaluation (Prefix Expression)

BEGIN:

Reverse (Prefix Expression)

STACK OperandStack

Initialize (OperandStack)

WHILE not end of input from postfix expression DO

Symbol $\leftarrow$ next character from prefix equation

```

        IF symbol is an operand THEN
            Push (OperandStack, symbol)
        ELSE
            oprnd 1 ← Push(OperandStack)
            oprnd 2 ← Push(OperandStack)
            value ← Result of applying symbol to oprnd1 and oprnd2
            Push(OperandStack, value)
            Result ← Pop(OperandStack)
        RETURN Result
    END;

```

Time Complexity: $\Theta(N)$

There are N symbols in the Expression. For each symbol decision is to be taken for Push or Pop. In case of operand 1 statement execution and for operator 4. There are $N/2+1$ operands and $N/2-1$ operators.

Hence total statements = Reverse + Initialization + $(N/2+1) * 1 + (N/2-1) * 3 +$ Return result

$$= C*N+1+2*N-2+2$$

$$= (C+2)*N+1$$

Space Complexity: $\Theta(N)$

$N/2+1$ size stack is required and three variables for Symbol, value, and result. For reversing the string another N size stack is required. Total space = $N+N/2+1+3=3N/2+4$

C-Program

```

/*****/
void main()
{
    InitialiseStack();
    int answer;
    char prefix[ ]="+-^234*63";
    printf("Prefix Expression\n");
    puts(post);
    answer=PrefixEvaluation(prefix);
}

```

```

    printf("value = %d", answer);
}
/*****/
int PrefixEvaluation(char prefix[ ])
{
    int i=0, symbol, x, op1, op2, value;
    strrev(post);
    while(prefix[i]!='\0')
    {
        symbol=prefix[i];
        if(symbol >= '0' && symbol <= '9')
            Push(symbol-'0');
        else
        {
            op1=Pop();
            op2=Pop();
            value=evaluation(op1, op2, symbol);
            Push(value);
        }
        i++;
    }
    x=Pop();
    return x;
}
/*****OUTPUT*****/
Prefix Expression
+-^234*63
value = 22
/*****/

```

10.3.5.2. Infix to Postfix Conversion

To convert the given Postfix expression to Postfix, Let us re WRITE the Precedence and Associativity rules. We are assuming that there are only Exponentiation ($\uparrow$), Division ($/$), Multiplication ($*$), Addition ($+$) and Subtraction ($-$) operators.

1. Precedence and Associativity Rules

Operators	Priority	Associativity
↑	Highest Priority	Right to Left
*	Second Highest Priority	Left to Right
/	Second Highest Priority	Left to Right
+	Lowest Priority	Left to Right
−	Lowest Priority	Left to Right

To realize this in the Conversion, Let us take a function named Precedence(a, b) which results the following.

Result	Condition
TRUE	a has higher precedence than b
TRUE	a has equal precedence than b and a&b are Left Associative
FALSE	a has equal precedence than b and a&b are Right Associative
FALSE	b has higher precedence than a

Call of Function	Result
Precedence(+, +)	TRUE
Precedence(+, -)	TRUE
Precedence(-, +)	TRUE
Precedence(-, -)	TRUE
Precedence(*, *)	TRUE
Precedence(/, /)	TRUE
Precedence(*, /)	TRUE
Precedence(↑, ↑)	FALSE
Precedence(↑, /)	TRUE
Precedence(↑, +)	TRUE
Precedence(+, ↑)	FALSE
Precedence(*, ↑)	FALSE

Consider an Infix Expression. To convert this to Postfix, one symbol from expression is taken at a time. Following rules are used for conversion

- Take an empty operator stack;
- If the symbol is an operand, add the symbol to postfix expression;
- If the symbol is the operator and stack is empty, push the symbol on stack;
- If the symbol is the operator and stack is not empty do the following.
 - Find Precedence (Stacktop item, Symbol). If the precedence is TRUE, Pop an item from stack and Add the symbol on Postfix Expression.

- If stack is not Empty after this Pop, Repeat the above statement again otherwise Push the Symbol on the Stack.
- If Precedence (Stacktop item, Symbol) is False, Push the symbol on the stack
- If all the symbols are finished, Pop the stack repeatedly and add symbols on the Postfix Expression.

\$ is treated as the symbol on end of expression

Consider an Infix Expression $A+B*C/D\uparrow E\uparrow F*G$

Symbol	Operator Stack	Postfix Expression	Precedence Function	
A		A		
+	+			
B		AB		
*	+, *		(+, *)	FALSE
C		ABC		
/		ABC*	(*, /)	TRUE
	+, /		(+, /)	FALSE
D		ABC*D		
$\uparrow$	+, /, $\uparrow$		(/, $\uparrow$)	FALSE
E		ABC*DE		
$\uparrow$	+, /, $\uparrow$, $\uparrow$		($\uparrow$, $\uparrow$)	FALSE
F		ABC*DEF		
*		ABC*DEF $\uparrow$	($\uparrow$, *)	TRUE
		ABC*DEF $\uparrow\uparrow$	($\uparrow$, *)	TRUE
		ABC*DEF $\uparrow\uparrow$ /	(/, *)	TRUE
	+, *		(+, *)	FALSE
G		ABC*DEF $\uparrow\uparrow$ /G		
\$		<i>ABC*DEF$\uparrow\uparrow$/G*+</i>		

ALGORITHM InfixToPostfix (Infix expression)

BEGIN:

STACK OperatorStack

Initialize (OperatorStack)

WHILE not the end of input from Infix Expression DO

Symbol $\leftarrow$ Next symbol from Infix Expression

IF Symbol is an operand THEN

Add symbol to postfix expression

ELSE

WHILE !Empty (OperatorStack) && Precedence (StackTop(OperatorStack) , Symbol) DO

$x \leftarrow \text{Pop}(\text{OperatorStack})$

 Add x to postfix Expression

 Push(OperatorStack, Symbol)

WHILE ! Empty(OperatorStack) DO

$x \leftarrow \text{Pop}(\text{OperatorStack})$

 Add x to Postfix Expression

 RETURN Postfix Expression

END;

Time Complexity: $\Theta(N)$

For any symbol, there are two decisions. If the symbol is an operand one statement is executed. Else 2 statements in case the loop condition is true otherwise 1 statement for Push. Overall there are the statements in the order of N.

Space Complexity: $\Theta(N)$

The maximum size of the stack at any point can be N

C-Program

```
/**
```

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
*/
```

```
#define SIZE 20
```

```
#define TRUE 1
```

```
#define FALSE 0
```

```
/**
```

```
void initialize();
```

```
void Push(char);
```

```
int Pop();
int StackTop();
int Empty();
int precedence(char, char);
void InfixToPostfix(char [ ]);
/*****/

struct
{
    char item[SIZE];
    int top;
}s;
/*****/

int main()
{
    int i;
    char Infix[100];
    printf("Enter the infix expression\n");
    gets(Infix);
    InfixToPostfix(Infix);
}
/*****/

int precedence(char a, char b)
{
    if(a == '^' || a == '*' || a == '/' || a == '%')
    {
        if(b == '^')
            return FALSE;
        else
            return TRUE;
    }
    else
    {
        if(a == '+' || a == '-')
        {
            return TRUE;
        }
    }
}
```

```

    if(b == '+'||b == '-')
        return TRUE;
    else
        return FALSE;
}
}
}

/*****/
void InfixToPostfix(char Infix[ ])
{
    int i=0, j=0, r;
    char Postfix[100];
    char symbol, d, x;
    initialize();
    while(Infix[i]!='\0')
    {
        symbol=Infix[i];
        if((symbol >= 48 && symbol <= 57) || (symbol >= 65 && symbol <= 90) ||
(symbol >= 97 && symbol <= 122))
// 0-9, A-Z, a-z
        {
            Postfix[j]=symbol;
            j++;
        }

    else
    {
        while((!Empty())&& precedence(StackTop(), symbol))
        {
            x=Pop();
            Postfix[j]=x;
            j++;
        }
        Push(symbol);
    }
}

```

```

}
i++;
}
while(!Empty())
{
    d=Pop();
    Postfix[j]=d;
    j++;
}
Postfix[j]='\0';
printf("postfix expression\n");
puts(Postfix);
}
/*****OUTPUT*****/
Enter the infix expression
a+b*c^d%e%f
postfix expression
abcd^*e%f%+
/*****/

```

10.3.5.3. Dealing with Infix Expression with Parentheses

- If a is the opening parenthesis (, precedence results false;
- If b is the opening parenthesis (, precedence results false;
- If b is the closing parenthesis), precedence results true;
- If a is opening parenthesis (& b is closing parenthesis), precedence results False (This is a special case false in which stack is Popped but Popped symbol is discarded).

Call of Function	Result
Precedence((, +)	FALSE
Precedence((, -)	FALSE
Precedence(-, ()	FALSE
Precedence(+, ()	FALSE
Precedence(*, ()	FALSE
Precedence(/,))	TRUE

Precedence(*,)	TRUE
Precedence(↑,)	TRUE
Precedence((,)	FALSE

Consider an Infix Expression

$A+(B*(C/D+E))$

Symbol	Operator Stack	Postfix Expression	Precedence Function	
A		A		
+	+			
(	+, (		(+,)	FALSE
B		AB		
*	+, (, *		((, *)	FALSE
(	+, (, *, (		(*,)	FALSE
C		ABC		
/	+, (, *, (, /		((, /)	FALSE
D		ABCD		
+		ABCD/	(/, +)	TRUE
	+, (, *, (, +		((, +)	FALSE
E		ABCD/E		
)		ABCD/E+	(+,)	TRUE
	+, (, *		((,))	FALSE
)		ABCD/E+*	(*,)	TRUE
	+		((,))	FALSE
\$		<i>ABCD/E+*+</i>		

ALGORITHM InfixToPostfix (Infix expression)

BEGIN:

STACK OperatorStack

Initialize (OperatorStack)

WHILE not the end of input from Infix Expression DO

Symbol ← Next symbol from Infix Expression

IF Symbol is an operand THEN

Add symbol to postfix expression

ELSE

WHILE ! Empty (OperatorStack) && Precedence (StackTop(OperatorStack) , Symbol) DO

$x \leftarrow \text{Pop}(\text{OperatorStack})$
 Add x to postfix Expression

IF Symbol == ')' THEN

$x \leftarrow \text{Pop}(\text{OperatorStack})$

ELSE

Push(OperatorStack, Symbol)

WHILE ! Empty(OperatorStack) DO

$x \leftarrow \text{Pop}(\text{OperatorStack})$

Add x to Postfix Expression

RETURN Postfix Expression

END;

The complexity of this Algorithm remains same as that of the conversion without parenthesis.

10.3.5.4. Infix to Prefix Conversion

Consider an Infix Expression. To convert this to Prefix, Following rules are used

- Reverse the infix expression.
- Take an empty operator stack.
- If the symbol is an operand, add the symbol to prefix expression.
- If the symbol is the operator and stack is empty, push the symbol on stack.
- If the symbol is the operator and stack is not Empty do the following:
 - Find Precedence(Symbol, StackTop item). If the precedence is FALSE, Pop an item from stack and Add the symbol on Prefix Expression.
 - If stack is not Empty after this Pop, Repeat the above statement again otherwise Push the Symbol on the Stack.
 - If Precedence (Stacktop item, Symbol) is True, Push the symbol on the stack
- If all the symbols are finished, pop the stack repeatedly and add symbols on the postfix expression.
- Reverse the prefix expression

Consider the Infix Expression $A+B*C/D \uparrow E \uparrow F * G$

Expression after reverse

$G * F \uparrow E \uparrow D / C * B + A$

Symbol	Operator Stack	Prefix Expression	Precedence Function	
G		G		
*	*			
F	*	GF		
↑	*, ↑	GF	(↑, *)	TRUE
E	*, ↑	GFE		
↑	*	GFE↑	(↑, ↑)	FALSE
	*, ↑		(↑, *)	TRUE
D	*, ↑	GFE↑D		
/	*	GFE↑D↑	(/, ↑)	FALSE
	*, /		(/, *)	TRUE
C		GFE↑D↑C		
*	*, /, *		(*, /)	TRUE
B		GFE↑D↑CB		
+	*, /	GFE↑D↑CB*	(+, *)	FALSE
	*	GFE↑D↑CB*/	(+, /)	FALSE
		GFE↑D↑CB*/*	(+, *)	FALSE
	+			
A		GFE↑D↑CB*/*A		
\$		GFE↑D↑CB*/*A+		
		+A**BC↑D↑EFG		

ALGORITHM InfixToPrefix (Infix expression)

BEGIN:

Reverse (Infix Expression)

STACK OperatorStack

Initialize (OperatorStack)

WHILE not the end of input from Infix Expression DO

Symbol ← Next symbol from Infix Expression

IF Symbol is an operand THEN

Add symbol to prefix expression

ELSE

WHILE ! Empty (OperatorStack) && ! Precedence (Symbol, StackTop(OperatorStack)) DO

x ← Pop (OperatorStack)

Add x to prefix Expression

Push(OperatorStack , Symbol)

```
WHILE ! Empty(OperatorStack) DO
    x ← Pop(OperatorStack)
    Add x to Prefix Expression
RETURN Reverse (Prefix Expression)
END;
```

Time Complexity: $\Theta(N)$

The reverse is done twice in the logic requiring $2*CN$ effort. For any symbol, there are two decisions. If the symbol is an operand one statement is executed. Else 2 statements in case the loop condition is true otherwise 1 statement for Push. Overall there are the statements in the order of N .

Space Complexity: $\Theta(N)$

The maximum size of the stack at any point can be N

C-Program

```
/*  
#include<stdio.h>  
#include<stdlib.h>  
#include<string.h>  
#define SIZE 20  
#define TRUE 1  
#define FALSE 0  
*/  
void initialize();  
void Push(char);  
int Pop();  
int StackTop();  
int Empty();  
int precedence(char, char);  
void InfixToPrefix(char [ ]);
```



```
/******/  
struct  
{  
    char item[SIZE];  
    int top;  
}s;  
/******/  
int main()  
{  
    int i;  
    Char Infix[100];  
    printf("Enter the infix expression\n");  
    gets(Infix);  
    InfixToPrefix(Infix);  
}  
/******/  
int precedence(char a, char b)  
{  
    if(a == '^' || a == '*' || a == '/' || a == '%')  
    {  
        if(b == '^')  
            return FALSE;  
        else  
            return TRUE;  
    }  
    else  
    {  
        if(a == '+' || a == '-')  
        {  
            if(b == '+' || b == '-')  
                return TRUE;  
            else  
                return FALSE;  
        }  
    }  
}
```

```
    }
}
/*****/
void InfixToPrefix(char Infix[ ])
{
    int i=0, j=0, r;
    char Prefix[100];
    char symbol, d, x;
    initialize();
    strrev(Infix);
    while(Infix[i]!='\0')
    {
        symbol=Infix[i];
        if((symbol >= 48 && symbol <= 57) || (symbol >= 65 && symbol <= 90) ||
(symbol >= 97 && symbol <= 122))
        {
            Prefix[j]=symbol;
            J++;
        }
        else
        {
            while((!Empty())&& !precedence(symbol, StackTop()))
            {
                x=Pop();
                Prefix[j]=x;
                J++;
            }
            Push(symbol);
        }
        i++;
    }
    while(!Empty())
    {
        d=Pop();
```

```
    Prefix[j]=d;
    j++;
}
Prefix[j]='\0';
printf("prefix expression\n");
strev(Prefix);
puts(Prefix);
}
/*****OUTPUT*****/
Enter the infix expression
a*b^c^d-e^f%g
prefix expression
-*a^b^cd%^efg
/*****/
```

EXERCISES

1. **Convert following Infix expression into Postfix expression using Tabular method.**
 $(a - b / c) * (d + (e * f) / g)$
2. **Solve the following:**
 - a) $((A - (B + C) * D) / (E + F))$ [Infix to postfix]
 - b) $(A + B) * C - (D - E) ^ F$ [Infix to prefix]
 - c) $7\ 5\ 2\ +\ /\ 4\ 1\ 5\ -\ /\ -$ [Evaluate the given postfix expression]
 - d) $A\ B\ C\ +\ /\ D\ E\ F\ -\ /\ -$ [Postfix to Infix]
 - e) $A\ B\ C\ +\ /\ D\ E\ F\ -\ /\ -$ [Evaluate the given postfix expression]
 $A=24, B=3, C=1, D=14, E=9, F=7$
 - f) $A - B / C + D * E + F$ [Infix to postfix]
 - g) $A * ((B + D) / E - F) * (G + H / I)$ [Infix to postfix]
3. **The Order followed by stack data structure is**
 - a. Random
 - b. FIFO
 - c. LIFO
 - d. None
4. **How many stack will be needed for the evaluation of a prefix expression**
 - a. 1
 - b. 2
 - c. 0
 - d. 3
5. **If the two stacks are implemented on a single array, the overflow occurs at**
 - a. $top1 = top2$
 - b. $top1 = top2 - 1$
 - c. $top1 - 1 = top2$
 - d. none of these
6. **If PRCD(x, y) is a function that returns TRUE is x has higher precedence than y or if x has the same precedence as that of y but left associative. A call for PRCD(\$, \$) will return (\$ is exponentiation)**
7. **The evaluated value of POSTFIX EXPRESSION $4\ 3\ *\ 4\ 2\ /\ -\ 3\ 2\ \$\ 2\ *\ +$ is**
8. **What is the reason for selection of prime no in the division method of hashing?**
.....
9. **The average number of probes required for successful search on Hash Table with 12 slots and Linear probing method for collision resolution with the following records is**
10. **Postfix Equivalent of $A\$B\$C\$D\E is**

- a. ABCDE\$\$\$\$ b. AB\$C\$D\$D\$E\$ c. abc\$\$DE\$\$ d. .
ABCD\$\$\$\$E\$

11. **What are the notations used in Evaluation of Arithmetic Expressions using prefix and postfix forms?**
12. **The stack can be used to find if the given string is palindrome or not. No of POP operations required to do the same on the string MISSISSIM is**
13. **Evaluated value of expression $- + * 4 2 9 / 6 2$ is**
14. **Which of the following applications may use a stack?**
 - a. A parentheses balancing program.
 - b. Keeping track of local variables at run time.
 - c. Syntax analyzer for a compiler.
 - d. All of the above.

15. Consider the following pseudocode:

```

declare a stack of characters
while ( there are more characters in the word to read )
{
    read a character
    push the character on the stack
}
while ( the stack is not empty )
{
    write the stack's top character to the screen
    pop a character off the stack
}

```

What is written to the screen for the input “carpets”?

- a. serc
- b. carpets
- c. steprac
- d. ccaarrppeeetss

16.

```

declare a character stack
while ( more input is available)
{
    read a character
    if ( the character is a '(' )

```

```

    push it on the stack
else if ( the character is a ')' and the stack is not empty )
    pop a character off the stack
else
    print "unbalanced" and exit
}
print "balanced"

```

Which of these unbalanced sequences does the above code think is balanced?

- A. ((())
 - B.))(())
 - C. (())(())
 - D. (())(())
17. Here is an infix expression: $4+3*(6*3-2)$. Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. What is the maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression?
- A. 1 B. 2 C. 3 D. 4 E. 5
18. For conversion of infix expression given below to postfix, no of PUSH and POP operation performed is.....
- $a * b + c / d \uparrow e \uparrow f$
19. The expression $a \uparrow b \uparrow c * d * e / f / g / h$ in polish notation is
20. Which among the following algorithm requires application of stacks (directly or indirectly)
- a. Quick sort
 - b. Heap sort
 - c. Selection sort
 - d. Bubble sort
21. PUSH and POP operations for evaluation of postfix expression $4\ 3\ 2\ \uparrow\ *\ 6 - 8 +$ is
- a. 9 PUSH, 8 POP
 - b. 5 PUSH, 4 POP
 - c. 5 PUSH, 9 POP
 - d. 9 PUSH, 9 POP

22. **Best suited data structure to find the validity of mathematical expression with all types of brackets e.g. [], { }, () is**
23. **Which among the following algorithms requires reversing the input string before processing through stacks**
 - a. Infix to postfix conversion
 - b. Infix to prefix conversion
 - c. Postfix evaluation
 - d. Prefix evaluation
24. **Let $\text{mulpop}(S, k)$ is an operation that removes k items from the stack S . In which of the following algorithms, the $\text{mulpop}()$ function can be applied?**
 - a. Infix to postfix conversion
 - b. Infix to prefix conversion
 - c. Postfix evaluation
 - d. None of these
25. **Let there be a stack of integer values. How many elements does the stack contain after following operations**

$\text{PUSH}(S, 1), \text{PUSH}(S, 2), \dots, \text{PUSH}(S, 10)$
 $\text{Mulpop}(S, 5)$
 $\text{PUSH}(S, 11), \text{PUSH}(S, 12), \dots, \text{PUSH}(S, 30)$
 $\text{Mulpop}(S, 10)$

 - a. 20
 - b. 30
 - c. 10
 - d. 15

Programming Question

- 1) **Write a Program in C for implementing 2 stack on a single Array**
- 2) **Write a Program in C for evaluation of infix expression**
- 3) **Write a Program in C for conversion of Logic Expression (include AND, OR, NOT, and XOR operators) to postfix**
- 4) **Write a Program in C for conversion of Logic Expression (include AND, OR, NOT, and XOR operators) to prefix**
- 5) **Write a Program in C for Evaluation of a Logic Expression (include AND, OR, NOT, and XOR operators) written in Postfix form.**

11

Recursion

CONTENTS

11.1. Concept Of Recursion	188
11.2. Application Problems Related To Recursion	188
11.3. Towers Of Hanoi.....	195
Exercises.....	199

11.1. CONCEPT OF RECURSION

Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. Several approaches of the problem solving have been developed keeping the Recursion in the Background. e.g.,

- Divide and conquer;
- Dynamic programming;
- Backtracking etc.

Base Condition: in the recursion is the condition on which the further reduction of the problem solving should stop. e.g.,

$$n! = n * (n-1)!$$

The reductions should stop when n becomes 0 and direct value for $0! = 1$ should be declared.

For understanding space complexity of Recursion, please go through Chapter 3 of this book wherein the concept of Activation Records are discussed.

Some functions related to the Recursion is designed in the next few pages.

11.2. APPLICATION PROBLEMS RELATED TO RECURSION

1. Program to Calculate the Factorial of a Number Using Recursion:

ALGORITHM FACTORIAL(a)

BEGIN :

IF $a = 0$ THEN

RETURN(1)

ELSE

RETURN($a * \text{FACTORIAL}(a-1)$)

END;

Time Complexity: $\Theta(N)$

Factorial function is called $N+1$ times. Each time activation record is Pushed onto stack. Upon the returns, the activation records are Popped. Total $N+1$ Push, $N+1$ Pop. Since Push and Pop requires constant time, the total time = $2(N+1) * C$ that can be written as $\Theta(N)$

Space Complexity: $\Theta(N)$

At any time the maximum pending activation records will be $N+1$ on call stack. Assuming Activation records to hold constant space, Space complexity of this Algorithm will be $\Theta(N)$

```
#include<stdio.h>
/*****FUNCTION DEFINITION*****/
int factorial(int a)
{
    if(a == 0)
        return 1;
    else
        return(a*factorial(a-1));
}
/*****OUTPUT*****/
main()
{
    int a, f;
    printf("Enter the number");
    scanf("%d", &a);
    f=factorial(a);
    printf("Factorial of a number=%d", f);
}
/*****OUTPUT*****/
Enter the number 5
Factorial of a number=120
/*****OUTPUT*****/
```

2. Write a Program to Calculate the Power of a Given Number Using Recursion:

$a^b = a * a^{b-1}$ (Recursive Process)

$a^0 = 1$ (Base Condition)

ALGORITHM POWER(a, b)

BEGIN:

```
IF b == 0 THEN
```

```
RETURN 1
```

```
ELSE
```

```
RETURN a*POWER(a, b-1)
```

```
END;
```

Time Complexity: $\Theta(N)$

Power function is called $N+1$ times (a^b to a^0). Each time activation record is Pushed onto stack. Upon the returns, the activation records are Popped. Total $N+1$ Push, $N+1$ Pop. Since Push and Pop requires constant time, the total time = $2(N+1) * C$ that can be written as $\Theta(N)$

Space Complexity: $\Theta(N)$

At any time the maximum pending activation records will be $N+1$ on call stack. Assuming Activation records to hold constant space, Space complexity of this Algorithm will be $\Theta(N)$

C-Program

```
#include<stdio.h>
```

```
/******FUNCTION DEFINITION*****
```

```
int power(int a, int b)
```

```
{
```

```
    if(b == 0)
```

```
        return 1;
```

```
    else
```

```
        return(a*power(a, b-1));
```

```
}
```

```
/******
```

```
main()
```

```
{
```

```
    int a, b, p;
```

```
    printf("Enter the numbers");
```

```
    scanf("%d%d", &a, &b);
```

```
    p=power(a, b);
```

```

    printf("power =%d", p);
}
/*****OUTPUT*****/
Enter the numbers 2 3
power =8
/*****/

```

3. Program to Calculate n Terms of Fibonacci Series Using Recursion:

Fibonacci (n) = 0 if n=1 (Base Condition)
 Fibonacci (n) = 1 if n=2 (Base Condition)
 Fibonacci (n) = Fibonacci (n-1) + Fibonacci (n-2) if n>2 (Recursive Definition)

ALGORITHM Fibonacci(a)

BEGIN:

IF a == 1 THEN

RETURN 0

ELSE

IF a == 2 THEN

RETURN 1

ELSE

RETURN Fibonacci(a-1) + Fibonacci(a-2)

END;

Time Complexity: $\Theta(2^N)$

Here, to evaluate the Fibonacci term of a number, two Fibonacci stacks are made.

Hence,

$T(N) = T(N-1) + T(N-2) + T(1)$. The Solution to this recursion results in $\Theta(2^N)$

Space Complexity: $\Theta(N)$

A call stack is formed for N, N-1, N-2.... 1 where the height doesn't exceed N at any point of time

C-Program

```

/*****/
#include<stdio.h>
/*****/
int fibonacci(int a)
{
    if(a == 1)
        return 0;
    else
    {
        if(a == 2)
            return 1;
        else
            return fibonacci(a-1)+fibonacci(a-2);
    }
}
/*****/
main()
{
    int n, f;
    printf("Enter the number ");
    scanf("%d", &n);
    f=fibonacci(n);
    printf("Result=%d", f);
}
/*****OUTPUT*****/
Enter the number 6
Result=5
/*****/

```

4. Program to Calculate Greatest Common Divisor (GCD/ HCF) of 2 Numbers:

GCD(a, b) = a

if a=b (Base Condition)

GCD(a, b) = GCD(a-b, b)

if a>b (Recursive Definition)

GCD(a, b) = GCD(a, b-a)

if a<b (Recursive Definition)

ALGORITHM GCD(a, b)**BEGIN:**IF $a == b$ THEN

RETURN a

ELSE IF $a > b$ THEN

RETURN GCD(a-b, b)

ELSE

RETURN GCD (a, b-a)

END;**Time Complexity:** $\Theta(\min(a, b))$

Here, the function calls itself for minimum of (a, b) times where each time function executes two statements

Space Complexity: $\Theta(N)$

The height of the call stack never exceeds N even if both the numbers are prime

C-Program

```
/*  
#include<stdio.h>  
*/  
int GCD(int a, int b)  
{  
    if(a == b)  
        return a;  
    else if(a > b)  
        return GCD(a-b, b);  
    else  
        return GCD(a, b-a);  
}
```

```

main()
{
    int a, b, p;
    printf("Enter the numbers");
    scanf("%d%d", &a, &b);
    p=GCD(a, b);
    printf("GCD =%d", p);
}
/*****OUTPUT*****/
Enter the numbers 4 6
GCD =2
/*****/

```

5. Program to Calculate Reverse of a Number Using Recursion:

ALGORITHM REV (a, len)

BEGIN:

IF len == 1 THEN

RETURN a

ELSE

RETURN((a%10)*pow(10, len-1))+REV(a/10, len-1)

END;

Time Complexity: $\Theta(N)$

In each call, it executes 2 statements i.e., comparison, and returning. There are **len** number of calls for reversing a number

Space Complexity: $\Theta(N)$

The height of the call stack is equal to the len of the number.

C-Program

```

/*****/
#include<math.h>
#include<stdio.h>
/*****/

```

```

int reverse(int a, int len)
{
    if(len == 1)
        return a;
    else
        return((a%10)*pow(10, len-1))+reverse(a/10, len-1);
}
/*****/

main()
{
    int a, b, p;
    printf("Enter the number");
    scanf("%d", &a);
    printf("Enter the length");
    scanf("%d", &b);
    p=reverse(a, b);
    printf("reverse =%d", p);
}
/*****OUTPUT*****/

Enter the number 321
Enter the length 3
reverse =123
/*****/

```

11.3. TOWERS OF HANOI

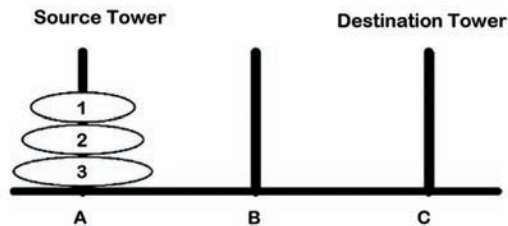
According to Hindu Mythology, A Rig-Veda a story says: Devtas and Danavs used to reside side by side wherein the Danavs used to create problems for Devtas. Devtas requested Lord Shiva to create the universe. Lord Shiva requested Lord Brahma the same. Lord Brahma created the universe with establishing three corners. Akash, Patal, and Dharti Lok. Akash Lok was given to Devtas, Patal to Danavs and Dharti to newly created creature the human beings. The first City created on the earth was Kashi. Lord Brahma established three towers in Akash, Patal, and Dharti Lok and 64 disks of reducing size (from bottom to top) on stack like arrangements at Tower in Akash Lok. The Devta were given the task for shifting of disks from Akash Lok tower to Patal Lok taking Dharti Lok tower as a mediator.

Story about an Indian temple Kashi Vishwanath in Varanasi is also of fame. This says the temple contains a big chamber with three time-worn posts in it, surrounded by 64 golden disks. Brahmin priests, acting out the command of an ancient insight, have been moving these disks in accordance with the immutable rules of Brahma since that time. The puzzle is therefore also known as the Tower of Brahma puzzle.

According to the legend, when the last move of the puzzle is completed, the world will end.

If the legend were true, and if the priests were able to move disks at a rate of one per day, using the smallest number of moves, it would take them $2^{64} - 1$ days.

The similar arrangement is made in the Buddhist temple in Hanoi (in Vietnam). The Monk make one move per day according to the rule that bigger disk cannot be kept above the smaller one.



Solution to Towers of Hanoi Problem

Towers of Hanoi is a classical recursion Problem whose solution can be written as

Transfer Disk from Source To Destination

if $n = 1$

Transfer $n-1$ disk from Source to Mediator

Transfer 1 disk from Source to Destination

Transfer $n-1$ disk from Mediator to Destination

} if $n > 1$

ALGORITHMTOH(N, S, M, D)

BEGIN:

IF $N = 1$ THEN

Transfer disk from S to D

ELSE

TOH(N-1, S, M, D)

Transfer Disk From S to D

TOH(N-1, M, S, D)

END;

Time Complexity: $\Theta(2^N)$

$$T(N) = 2T(N-1) + 1$$

$$T(N-1) = 2T(N-2) + 1$$

$$T(N) = 2(2T(N-2)+1)+1$$

.....

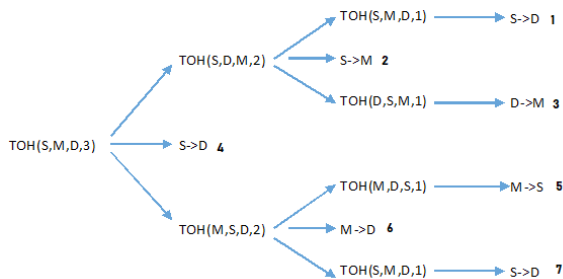
$$T(N) = 2^N T(0) + 2^{N-1} + 2^{N-2} + \dots 2^1 + 1$$

$$T(N) = 2^N$$

Space Complexity: $\Theta(N)$

For transferring the disk, a call stack of order N is formed

Run of Towers of Hanoi for 3 Disks



$$\text{Total Movements} = 2^3 - 1 = 7$$

C-Program

```

/*****
#include<stdio.h>
/*****
void TowersOfHanoi(int, char, char, char);
void main()
{
    int n;
    printf("enter the value");
    scanf("%d", &n);
    TowersOfHanoi(n, 'S', 'M', 'D');
}
/*****/

```

```
void TowersOfHanoi(int n, char s, char m, char d)
{
    if(n == 1)
    {
        printf("Transfer disk from %c to %c\n", s, d);
        return;
    }
    else
    {
        TowersOfHanoi(n-1, s, d, m);
        printf("Transfer disk from %c to %c\n", s, d);
        TowersOfHanoi(n-1, m, s, d);
    }
}

/*****OUTPUT*****/
enter the value 4
Transfer disk from S to M
Transfer disk from S to D
Transfer disk from M to D
Transfer disk from S to M
Transfer disk from D to S
Transfer disk from D to M
Transfer disk from S to M
Transfer disk from S to D
Transfer disk from M to D
Transfer disk from M to S
Transfer disk from D to S
Transfer disk from M to D
Transfer disk from S to M
Transfer disk from S to D
Transfer disk from M to D
/*****/
```

EXERCISES

Multiple Choice Questions

- 1) **Recursion is a method in which the solution of a problem depends on _____**
 - a) Larger instances of different problems
 - b) Larger instances of the same problem
 - c) Smaller instances of the same problem
 - d) Smaller instances of different problems
- 2) **Which of the following problems can be solved using recursion?**
 - a) Factorial of a number
 - b) Nth Fibonacci number
 - c) Length of a string
 - d) All of the mentioned
- 3) **Recursion is similar to which of the following?**
 - a) Switch Case
 - b) Loop
 - c) If-else
 - d) None of the mentioned
- 4) **In recursion, the condition for which the function will stop calling itself is _____**
 - a) Best case
 - b) Worst case
 - c) Base case
 - d) There is no such condition
- 5) **Consider the following code snippet:**

```
void my_recursive_function()
{
my_recursive_function();
}
int main()
{
my_recursive_function();
return 0;
}
```

What will happen when the above snippet is executed?

- a) The code will be executed successfully and no output will be generated
- b) The code will be executed successfully and random output will be generated
- c) The code will show a compile time error
- d) The code will run for some time and stop when the stack overflows

6) What is the output of the following code?

```
void my_recursive_function(int n)
{
if(n == 0)
return;
printf("%d ,", n);
my_recursive_function(n-1);
}
int main()
{
my_recursive_function(10);
return 0;
}
```

- a) 10
- b) 1
- c) 10 9 8...1 0
- d) 10 9 8...1

7) What is the base case for the following code?

```
void my_recursive_function(int n)
{
if(n == 0)
return;
printf("%d ,", n);
my_recursive_function(n-1);
}
int main()
{
```

```
my_recursive_function(10);  
return 0;  
}
```

- a) return
- b) printf(“%d , ” n)
- c) if(n == 0)
- d) my_recursive_function(n-1)

8) How many times is the recursive function called, when the following code is executed?

```
void my_recursive_function(int n)  
{  
if(n == 0)  
return;  
printf(“%d , ”n);  
my_recursive_function(n-1);  
}  
int main()  
{  
my_recursive_function(10);  
return 0;  
}
```

- a) 9
- b) 10
- c) 11
- d) 12

9) What does the following recursive code do?

```
advertisement  
void my_recursive_function(int n)  
{  
if(n == 0)  
return;  
my_recursive_function(n-1);  
printf(“%d , ”n);
```

```

}
int main()
{
my_recursive_function(10);
return 0;
}

```

- a) Prints the numbers from 10 to 1
- b) Prints the numbers from 10 to 0
- c) Prints the numbers from 1 to 10
- d) Prints the numbers from 0 to 10

10) Which of the following statements is true?

- a) Recursion is always better than iteration
- b) Recursion uses more memory compared to iteration
- c) Recursion uses less memory compared to iteration
- d) Iteration is always better and simpler than recursion

11. Pick the odd one out

- a. Random Value
- b. Return Address
- c. Local variable space
- d. Global variable space

12. The function for which among the following will not be linear recursion

- a. Factorial
- b. Fibonacci Series
- c. a^b
- d. sum of the natural numbers

13. No. of steps needed for transfer of n disks from Tower A to Tower B using Tower C in the Tower of Hanoi problem is

- a. 2^n
- b. $2^n - 1$
- c. 2^{n+1}
- d. 2^{n-1}

14. How many calls for f(3) will be made for the following recurrence in the call of f(8)

$$f(n) = \begin{matrix} 0 & n=1 \\ 1 & n=2 \\ f(n-1)+f(n-2) & n>2 \end{matrix}$$

- a. 7
- b. 6
- c. 5
- d. 8

Let m and n be the integers and $A(m, n)$ be a recursively defined function

$$A(m, n) = \begin{cases} 5 & \text{if } m < n \\ A(m-n, n+2) + m & \text{if } m \geq n \end{cases}$$

- 15. What is the value of $A(2, 7)$**
- 2
 - 7
 - 5
 - none of these
- 16. What is the value of $A(5, 3)$**
- 5
 - 3
 - 8
 - 10
- 17. What is the value of $A(15, 2)$**
- 15
 - 2
 - 5
 - 42

Let $\text{Gcd}(m, n)$ be a recursively defined function as given below

$$\text{Gcd}(n, m) = \begin{cases} m < n \\ \text{Gcd}(m, n) = m & n=0 \\ \text{Gcd}(n, m \% n) & \text{otherwise} \end{cases}$$

- 18. $\text{Gcd}(6, 15) = ?$**
- 6
 - 15
 - 3
 - 0
- 19. $\text{Gcd}(20, 28) = ?$**
- 4
 - 20
 - 7
 - 28
- 20. $\text{Gcd}(18, 60) = ?$**
- 9
 - 3
 - 6

- d. none of these

Programming Questions

- 1) **Write a Program in C for computing ab using divide and conquer.**
- 2) **Write a Program in C for putting four queens in board on size 4x4 such that no queen is in attacking position.**

12

Queue

CONTENTS

12.1. Queue Basics.....	206
12.2. Linear Queue.....	206
12.3. Circular Queue.....	211
12.4. Priority Queues.....	217
Exercises.....	222

12.1. QUEUE BASICS

The queue is a linear collection of data items. They follow the principle of first-in-first-out (FIFO), which means that the first element added in the queue would be the first one to be removed. There are two ends in a Queue: REAR and FRONT. Insertions take place at the REAR end and Deletions takes place from the FRON-end.

The queues are used wherever we have a speed mismatch between the two entities out of which one is submitting the requests and other one handling those requests. e.g.,

- Input-output buffers.
- Printer buffers.
- CPU scheduling.
- Real-time booking systems, like flash sales and ticket booking systems.
- Round robin scheduling etc.
- In real life, we can relate it to queue in FRONT of the ticket window that offers tickets to the one who arrived first and in the same order as they arrived. The service here is in a first come first serve basis (FCFS).
- Automatic signaling at railway tracks.
- Packing of manufactured items at conveyer belt in a factory.
- It is used for implementation of various algorithms like breadth-first search (BFS) in the graph, level order traversal in tree, etc.
- Delivery of messages.

Types of queues:

Linear queue;

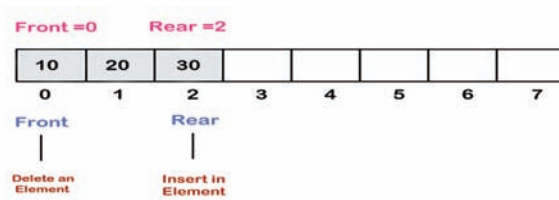
Circular queue;

Double ended queue;

Priority queue.

12.2. LINEAR QUEUE

The linear queue is used in the scenario where the queue has to be used for one-time use only.



- **Primitive Operations on Queue:** Here, it is assumed that Queue is implemented using Array.
- **Initialization:** REAR and FRONT is initialized to show that there are zero elements on the queue initially. REAR is initialized to -1 and FRONT at 0. At any point of time, the total number of Elements in the queue can be given by the formula:

$$\text{REAR} - \text{FRONT} + 1$$

Initially, $-1 - 0 + 1 = 0$ Elements are there in the queue

ALGORITHM Initialize(Queue Q)

BEGIN:

Q.REAR $\leftarrow$ -1

Q.FRONT $\leftarrow$ 0

END;

Time Complexity: $\theta(1)$

Only two statements in the Algorithm

Space Complexity: $\theta(1)$

No Extra variable taken

1. **Insertion in Queue:** Insertion is given a standard name Enqueue. Insertions take place at REAR end. If REAR has reached to Extreme end of the Array, further Insertion is not possible. An attempt to insert an element further may lead to overflow condition.

ALGORITHM EnQueue(Queue Q, key)

BEGIN:

IF Q.REAR == SIZE-1 THEN

WRITE ("Queue Overflows")

EXIT (1)

ELSE

Q.REAR $\leftarrow$ Q.REAR+1

Q.Item[Q.REAR] $\leftarrow$ key

END;

Time Complexity: $\theta(1)$

In adding an element to the queue, 3 statements are executed

Space Complexity: $\theta(1)$

No extra variable is used here

2. **Empty Check:** This is a Boolean value function that returns true or false depending on if the queue contains zero element or not

ALGORITHM Empty (Queue Q)

BEGIN:

IF $Q.REAR - Q.FRONT + 1 == 0$ THEN

RETURN TRUE

ELSE

RETURN FALSE

END;

3. **Deletions in Queue:** Deletion is given a standard name Dequeue. Deletion takes place from $FRONT$ end. If there are zero elements in the Array, further Deletion will not be possible. An attempt to Delete an element further may lead to underflow condition.

ALGORITHM DeQueue (Queue Q)

BEGIN:

IF $Q.REAR - Q.FRONT + 1 == 0$ THEN

WRITE("Queue Underflows")

EXIT(1)

ELSE

$X \leftarrow Q.item[Q.FRONT]$

$Q.FRONT \leftarrow Q.FRONT + 1$

RETURN X

END;

Time Complexity: $\theta(1)$

In deleting an element to the queue, 2/3 statements are executed conditionally

Space Complexity: $\theta(1)$

Only one extra variable is used, i.e., X

4. **Limitation of Linear Queue:** In case the REAR has reached to the last index, i.e., the extreme right in the array, further insertion will not

be possible even if there are free slots on feet side. To overcome this, Circular Queue is used wherein after insertion at last index, REAR is moved to the beginning of the array.

C-Program

```

/*****/
#include<stdio.h>
#include<stdlib.h>
#define SIZE 10
/*****/

struct Queue
{
    int REAR;
    int FRONT;
    int item[SIZE];
}q;
/*****/

void Intialization()
{
    q.REAR ← -1;
    q.FRONT ← 0;
}
/*****/

void EnQueue(int key)
{
    if (q.REAR == SIZE)
    {
        printf("Queue Overflows");
        exit(1);
    }
    q.REAR++;
    q.item[q.REAR]=key;
}

```

```
/******  
int DeQueue()  
{  
    int x;  
    if ((q.REAR-q.FRONT+1) == 0)  
    {  
        printf("Queue Underflows");  
        exit(1);  
    }  
    x=q.item[q.FRONT];  
    q.FRONT++;  
    return x;  
}  
/******  
main()  
{  
    int x;  
    Initialization();  
    EnQueue('A');  
    EnQueue('B');  
    EnQueue('C');  
    x=DeQueue();  
    printf("%c\n", x);  
  
    EnQueue('S');  
    EnQueue('R');  
    EnQueue('D');  
    EnQueue('H');  
    EnQueue('G');  
    EnQueue('F');  
    x=DeQueue();  
    printf("%c\n", x);  
  
    x=DeQueue();
```

```

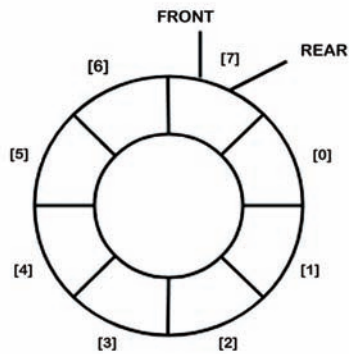
printf("%c\n", x);

x=DeQueue();
printf("%c\n", x);

x=DeQueue();
printf("%c\n", x);
}
/*****OUTPUT*****/
A
B
C
S
R
/*****/

```

12.3. CIRCULAR QUEUE



**A simple circular queue
with Size 8**

- 1. Initialization:** REAR and FRONT are initialized to show that there are zero elements on the queue initially. Both REAR and FRONT are initialized to SIZE-1 index.

ALGORITHM Initialize(CQueue CQ)

BEGIN:

CQ.REAR $\leftarrow$ SIZE-1

CQ.FRONT $\leftarrow$ SIZE-1

END;

Time Complexity: $\theta(1)$

Only 2 statements in the Algorithm

Space Complexity: $\theta(1)$

No Extra variable taken

- 2. Empty Check:** This is a Boolean value function that returns true or false depending on if the queue contains zero element or not. The Circular Queue will be Empty if REAR is equal to FRONT.

ALGORITHM Empty (Queue Q)

BEGIN:

IF CQ.REAR == CQ.FRONT THEN

RETURN TRUE

ELSE

RETURN FALSE

END;

- 3. Deletion in Queue:** Deletion is given a standard name i.e., DeQueue. Deletions take place from FRONTend. If there are zero elements in the Queue, deletion will not be possible. An attempt to delete an element further may lead to underflow condition.

In Linear Queue, the deletion from FRONTend is performed then the FRONT is updated. In Circular Queue first FRONT is updated then the Deletion is performed.

ALGORITHM DeQueue (Queue Q)

BEGIN:

IF Empty(CQ) THEN

WRITE("Queue Underflows")

EXIT(1)

ELSE

CQ.FRONT $\leftarrow$ (CQ.FRONT+1)%SIZE

```
X ← CQ.Item[CQ.FRONT]
RETURN X
END;
```

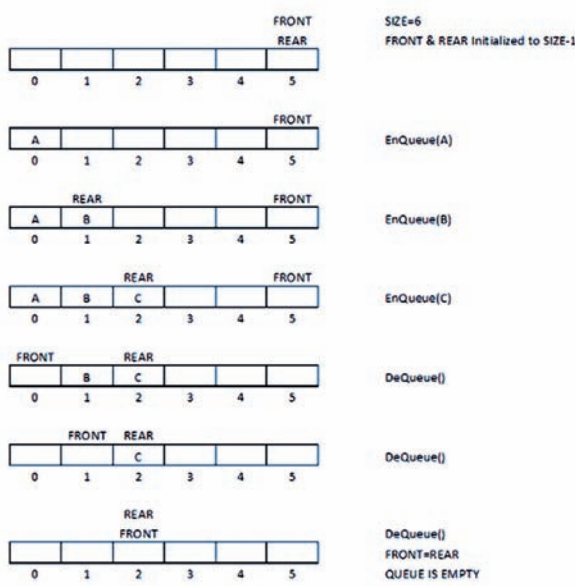
Time Complexity: $\theta(1)$

In deleting an element from the queue, 2/3 statements are executed conditionally

Space Complexity: $\theta(1)$

Only one extra variable is used i.e., x

4. **Insertion in Queue:** Insertion is given a standard name i.e., EnQueue. Insertions take place at the REARend. For Insertion of an element in the Circular Queue, First REAR is updated. If by updating the REAR it becomes equal to FRONT, Insertionsare not permitted. In case we allow the Insertion, REAR will become equal to FRONT. If we will try to Delete an element later to this insertion, it will perceive that Circular queue is Empty even though the Queue is Full. To avoid this condition, We restrict the Queue Insertion when Queue has SIZE –1 elements. An attempt to Insert an element further may lead to overflow condition.



```
ALGORITHM EnQueue(CQueue CQ, key)
BEGIN:
```

```

IF (CQ.REAR+1)%SIZE == CQ.FRONT THEN
WRITE ("Queue Overflows")
EXIT (1)
ELSE
CQ.REAR ← (CQ.REAR+1)%SIZE
CQ.Item [CQ.REAR] ← key
END;

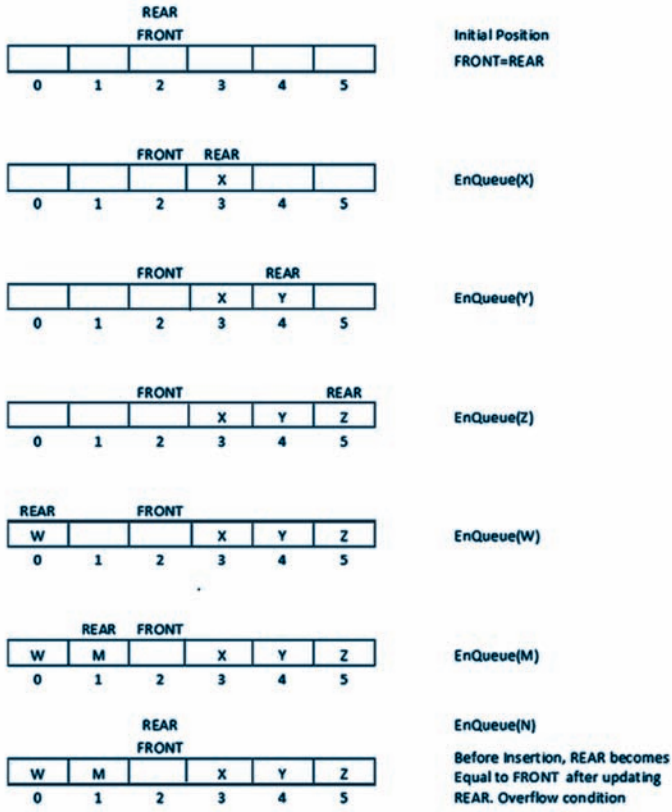
```

Time Complexity: $\Theta(1)$

In adding an element to the queue, 3 statements are executed

Space Complexity: $\Theta(1)$

No extra variable is used here



C-Program

/* */

#include<stdio.h>

```
#include<stdlib.h>
#define SIZE 10
/*****/
void Intialize();
void EnQueue(char);
int DeQueue();
/*****/
struct CircularQueue
{
    int REAR;
    int FRONT;
    char item[SIZE];
}cq;
/*****/
main()
{
    char x;
    Intialize();
    EnQueue('A');
    EnQueue('B');
    EnQueue('C');
    x= DeQueue();
    printf("%c", x);

    EnQueue('Q');
    EnQueue('W');
    EnQueue('E');
    EnQueue('T');
    EnQueue('Y');
    x= DeQueue();
    printf("\n%c", x);

    x= DeQueue();
    printf("\n%c", x);
```

```

x= DeQueue();
printf("\n%c", x);

x= DeQueue();
printf("\n%c", x);
}
/*****/
Intialize()
{
    cq.REAR=SIZE-1;
    cq.FRONT=SIZE-1;
}
/*****/
EnQueue(char key)
{
    if(((cq.REAR+1)%SIZE) == cq.FRONT)
    {
        printf("Queue Overflows");
        exit(1);
    }
    cq.REAR=(cq.REAR+1)%SIZE;
    cq.item[cq.REAR]=key;
}
/*****/
int DeQueue()
{
    char x;
    if(cq.REAR == cq.FRONT)
    {
        printf("Queue Underflows");
        exit(1);
    }
    cq.FRONT=(cq.FRONT+1)%SIZE;

```

```

x=cq.item[cq.FRONT];
return x;
}
/*****OUTPUT*****/
A
B
C
Q
W
/*****/

```

12.4. PRIORITY QUEUES

Here, each element has its own priority. For example, sorting integer elements, the priority of smaller elements will be more than the larger elements. While Execution of the jobs at CPU, some of the processes have higher priority than others. These processes are given the chance of execution at CPU prior to low priority processes.

It is most widely used in:

- Creating sorted sequences like in bandwidth management.
- It is used for implementation of Dijkstra's Algorithm, Minimal Spanning Tree Prims Algorithm, Huffman Coding, etc.
- CPU Scheduling.
- A*search Algorithm in Artificial Intelligence.

1. **Insertion in Priority Queue:** Insertions in the priority queue are done such that higher priority elements get the appropriate position.
2. **Deletions in Priority Queue:** Highest Priority element is Deleted First. Higher priority elements get deleted before lower priority elements. If two elements are of the same priority, they are deleted in FIFO order.
3. **Implementation of Priority Queue:** Priority Queue can be implemented using Array, Heap or Linked List

12.4.1. Array Implementation of Priority Queue

For Array implementation of Priority Queue, it is assumed that the element content denote the Priority. Lower numbers denote higher priority and higher number denote lower priority.

1. Insertion in Priority Queue**ALGORITHM PQInsert(A[], N, key)****BEGIN:** $i \leftarrow 0$ **WHILE** $i \leq N-1$ **AND** $\text{key} \geq A[i]$ **DO** $i \leftarrow i+1$ **FOR** $j \leftarrow N-1$ **TO** i **STEP-1 DO** $A[j+1] \leftarrow A[j]$ $A[i] \leftarrow \text{key}$ $N \leftarrow N+1$ **END;****Time Complexity:** $\theta(N)$

The while loop runs i number of statements, the for loop runs $n-i$ statements and three statements for assigning the value of i , $A[i]$, and incrementing the value of N . In any case, $i+n-i+3=n+3$ statements are executed.

Space Complexity: $\theta(1)$

Here, only two variables i and j are used.

ALGORITHM PQDeletion(A[], N)**BEGIN:** $x \leftarrow A[0]$ **FOR** $j \leftarrow 1$ **TO** N **DO** $A[j-1] \leftarrow A[j]$ $N \leftarrow N-1$ **RETURN** x **END;****C-Program**

```

/*****

```

```
#include <stdio.h>

/*****/

void Traverse(int A[ ], int N){
for (int i=0; i <= n-1; i++) {
printf(“%d”, A[i]);
}
}

/*****/

void PQInsertion(int A[ ], int *N, int key){
int i=0;
while (key >= A[i]) {
i=i+1;
}
for (int j=*N-1; j >= i; j--) {
A[j+1]=A[j];
}
A[i]=key;
*N=*N+1;
}

/*****/

int PQDelete(int A[ ], int *N)
{
int x;
x=A[0];
for (j=1; j <= *N-1; j++)
    A[j-1]=A[j]
*N=*N-1
RETURN x
}

/*****/

int main(int argc, constchar * argv[ ])
{
int A [20]={1, 2, 3, 4, 5, 6, 7};
int N=7;
```



```

PQInsertion(A, &N, 6);
Traverse(A, N);
x =PQDelete(A, &N)
printf(“Deleted Element from Priority Queue is:=> %d”, x);
}
/******

```

12.4.2. Heap Implementation

Heaps can be realized as Priority Queue. It is considered that key in the nodes in the heap represents Priority.

If repeated deletions in Max Heap are performed, it gives the numbers in Descending sequence. Hence it can be renamed as descending priority queue.

If repeated deletions in Min Heap are performed, it gives the numbers in ascending sequence. Hence it can be renamed as ascending priority queue

Insertions and Deletions in the Ascending Priority Queue (Min Heap are given As)

ALGORITHM PQInsert(A[], N, key)

BEGIN:

```

    A[N+1] ← key
    i ← N+1
    WHILE i>1 AND A[i]<A[i/2] DO
        Exchange(A[i], A[i/2])
    i ← i/2
    N ← N+1

```

END;

Time Complexity: $\Theta(\log_2 N)$

Space Complexity: $\Theta(1)$

ALGORITHM ExtractMin(A[], N)

BEGIN:

```

    key ← A[1]
    A[1] ← A[N]
    Adjust(A, 1, N-1)

```

$$N \leftarrow N - 1$$
END;**Time Complexity: $\Theta(\log_2 N)$**

When the element is removed from the top Adjust is called whose complexity is $\log_2 N$. Hence it is $3 + \log_2 N$

Space Complexity: $\Theta(1)$

The only new variable is key

EXERCISES

1. **Show the result of sequence of following operation on a Linear Queue of Size 8. Queue is Empty initially**
 - a. Two items A and B are inserted
 - b. One item is removed
 - c. Three items P, Q, R are inserted
 - d. One item is removed
 - e. Two items X, Y are inserted
 - f. One item is removed
 - g. One item C and D are inserted
2. **Show the result of sequence of following operation on a Circular Queue of Size 8. Queue is Empty initially**
 - a. Two items A and B are inserted
 - b. One item is removed
 - c. Three items P, Q, R are inserted
 - d. One item is removed
 - e. Two items X, Y are inserted
 - f. One item is removed
 - g. One item C and D are inserted
3. **Write a C Program for implementation of Input restricted Double ended queue**
4. **Write a C Program for implementation of Output restricted Double ended queue**
5. **Consider the double ended queue given below**

Right						Left			
A	B	C					D	E	F

The overflow occurs when

- a. Left = Right
 - b. Left + 1 = Right
 - c. Left = right + 1
 - d. None
6. **Given a linear queue**

Front			Rear			
1	2	3	4	5	6	7
		A	B	C		

After the following operations, what is the position of front and rear?

Insert(Q, D) Remove(Q) Insert(Q, E)

- Rear=6, Front=3
- Rear=7, Front=4
- Rear=7, Front=3
- None of these

7. Given a linear queue

Front			Rear			
1	2	3	4	5	6	7
		A	B	C		

After the following operations, what is the position of front and rear?

Insert(Q, D) Insert(Q, E) Remove() Insert(Q, F)

- Rear=7, Front=4
- Rear=7, Front=3
- Rear=8, Front=3
- Queue overflows

8. Given a circular queue

Rear				Front		
1	2	3	4	5	6	7
A	B	C				

After the following operations, what is the position of front and rear?

Insert(Q, X), Insert(Q, Y), Remove(Q), Insert(Q, Z)

- Rear=6, Front=1
- Rear=6, Front=7
- Rear=7, Front=1
- None of these

9. Given a circular queue

Rear				Front		
1	2	3	4	5	6	7
A	B	C				

After the following operations, what is the position of front and rear?

Insert(Q, X), Insert(Q, Y), Remove(Q), Insert(Q, Z) Insert(Q, D), Insert(Q, E)

- Rear=1, front=1

- b. Rear=1, front=7
 - c. Rear=1, front=2
 - d. Queue overflows
- 10. When linear queue of size MAXQUEUE were implemented in C, underflow occurs at**
- a. Front = 0
 - b. Rear = -1
 - c. Rear = Front
 - d. $\text{Rear} - \text{Front} + 1 = 0$
- 11. When linear queue of size MAXQUEUE were implemented in C, overflow occurs at**
- a. Front = MAXQUEUE
 - b. Rear = MAXQUEUE -1
 - c. Rear = Front
 - d. $\text{Rear} - \text{Front} + 1 = 0$
- 12. In a circular queue of size MAXQUEUE, overflow occurs at**
- a. Before incrementing rear, rear = front
 - b. After incrementing rear, rear = front
 - c. Before incrementing rear, rear = front +1
 - d. After incrementing rear, rear = front +1
- 13. In a circular queue of size MAXQUEUE, overflow occurs at**
- a. Before incrementing rear, rear = front
 - b. After incrementing rear, rear = front
 - c. Before incrementing rear, rear = front +1
 - d. After incrementing rear, rear = front +1

13

Linked List

CONTENTS

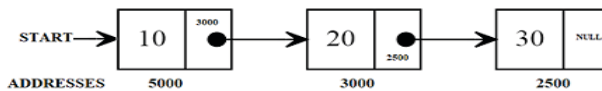
13.1. Linked List Basics.....	226
13.2. Linear Linked List.....	227
13.3. Related Functions	237
13.4. Set Operations Using Linked List	258
13.5. Polynomial Representation And Polynomial Addition	267
13.6. Long Numbers Arithmetic.....	273
13.7. Linked List Implementation Of Queue	275
13.8. Linked List Implementation Of Stack.....	280
13.9. Linked List Implementation Of Priority Queue	286
13.10. Circular Linked List.....	290
13.11. Doubly Linked List.....	302
13.12. Circular Doubly Linked List	318
Exercises.....	332

13.1. LINKED LIST BASICS

It is a linear collection of data elements. Here, their order is kept on record by dynamically storing the address of the next element instead of linear placements of elements and storing the address of the first element. The benefit of linked lists is that the size of the list can be changed until the entire memory in the process boundary has been utilized. Insertion and Deletion of elements is less costly in linked list than in array.

They are the most basic data structure that can be used to implement almost every other data structures including stacks, queues, graphs, trees, heaps, etc.

Linked List is the collection of nodes. Each node contains at least two fields. One field contains the data element, and the other one contains the address of the next node.



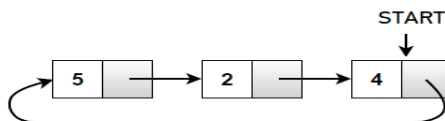
START keeps the address of the first node. Next field of the last node contains NULL

Linked lists are used by the system at various levels like:

- Dynamic partitioning during memory management.
- Keeping track of free and allocated disk blocks.
- Maintaining the process control block.
- Polynomial arithmetic.
- Long number arithmetic.
- Implementation of stack, queue, priority queue, tree, graph, tries, etc.

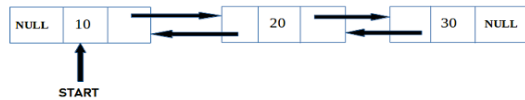
There are various types of linked lists

- **Linear Linked List:** Above diagram shows Linear Linked List where each node contains the address of next node and last node contains NULL address in the next field.
- **Circular Linked List:** Circular Linked List is more like Linear Linked List except that the next field of the last node contains the address of the first node.

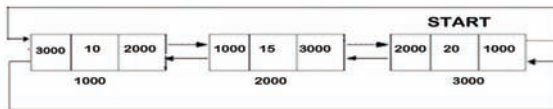


- **Doubly Linked List:** Each node in this type of the linked list contains three fields. One contains information, one is used for address of left

node and another one is used for storing the address of the right node. Left field of the first node and Right field of the last node contains NULL. Binary Trees are implemented using the doubly linked list.



- Circular Doubly Linked List:** As in Doubly, Each node in this type of the linked list contain three fields. One contains information, one is used for address of previous node and another one is used for storing the address of the next node. Left field of first node keeps the address of last node and right field of the last node contains the address of first node. START keeps the address of the last node as in circular linked list.



13.2. LINEAR LINKED LIST

In linear Linked List, there are two fields in a node.

P	
Info	Next

If P is the address of the Node, the fields are accessed as:

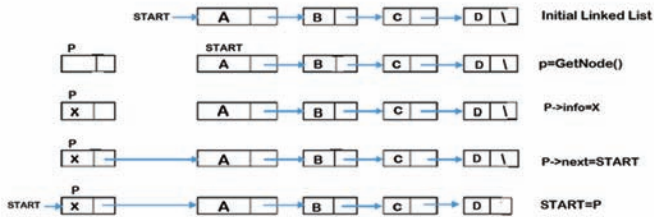
P → Info

P → Next

13.2.1. Primitive Operations

- Initialization:** For initialization, START is set NULL
- Insertions:** Insertions can be performed at the BEGINning, somewhere in between or in the end. The below-mentioned algorithms explain the Insertion process.
- Insertion at the Beginning:** For Insertion at the Beginning, a new node is taken. Let us assume whenever we required a new node, Call to GetNode will provide us with that.

After Linking the new node with the first node, START, that keeps the address of the first node, is required to be shifted to new node.



ALGORITHM InsertBeginning(START, key)

BEGIN:

$p = \text{GetNode}()$

$p \rightarrow \text{Info} = \text{key}$

$p \rightarrow \text{Next} = \text{START}$

$\text{START} = p$

END;

Time Complexity: $\theta(1)$

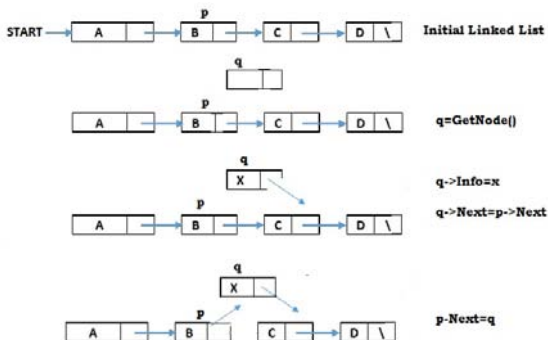
The number executed statements are 4 in the above algorithm.

Space Complexity: $\theta(1)$

An extra variable p is used. Also, the space is allocated in the memory to the new node.

Note $\rightarrow$ Since Linked list Algorithms will require the $\rightarrow$ operator frequently, in place of $\leftarrow$ we are using $=$ for assignment just for not to get confused. The same convention is used in Binary Tree Algorithms also.

- **Insertion After the Given Node:** For this operation, A new Node needs to be inserted in between the current node and node next to it.



ALGORITHM InsertAfter(p, key)**BEGIN:**

q=GetNode()

q→ Info=key

q→Next=p→ Next

p→Next=q

END;**Time Complexity:** $\theta(1)$

The number executed statements are 4 in the above algorithm.

Space Complexity: $\theta(1)$

An extra variable **q** is used. Also, the space is allocated in the memory to the new node.

- **Insertion at the End:** For Insertion at the end, Linked list is required to be Traversed till the last node. The New Node is inserted after the last node by setting up the pointer. Next field of the newly created node should be set to NULL as this will be the new last Node.

ALGORITHM InsertEnd(START, key)**BEGIN:**

p=START

WHILE p!=NULL DO

p=p→ Next

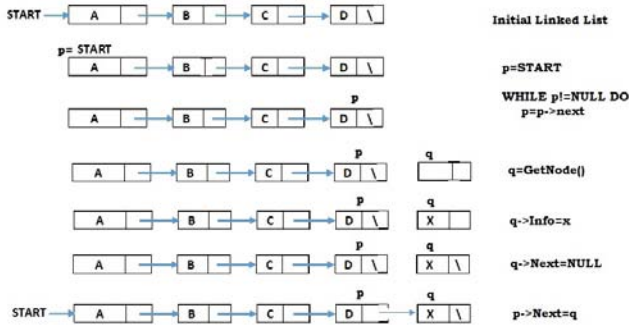
q = GetNode()

q→Info=key

q→ Next=NULL

p→ Next=q

END;



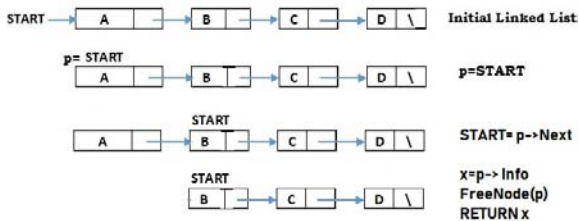
Time Complexity: $\Theta(N)$

The while loop runs until the end of the linked list is reached. Hence, for a linked list of N items, $N+5$ statements will be executed

Space Complexity: $\Theta(1)$

Extra variables used here are **p** and **q**. Also, the space is allocated in the memory to the new node

- **Deletions:** For performing Deletions, The memory occupied by the node to be deleted should be destroyed and information content should be returned to the calling function. Let us assume that FreeNode Function destroys memory for the given node.
- **Deletions from the Beginning:** For Deletion of the First node, The START is required to be shifted to the second node.



ALGORITHM DelBeg(START)

BEGIN:

$p = \text{START}$

$\text{START} = \text{START} \rightarrow \text{Next}$

$x = p \rightarrow \text{Info}$

$\text{FreeNode}(p)$

RETURN x
END;

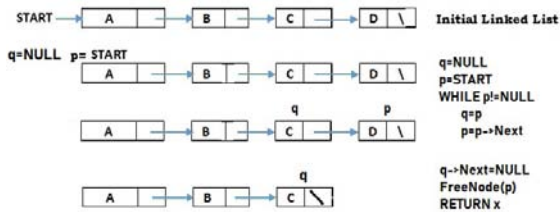
Time Complexity: $\theta(1)$

The above algorithm executes four statements unconditionally

Space Complexity: $\theta(1)$

Two extra variables, p & x are used

- Deletions from the END:** For Deletion from the end, linked list is required to be traversed till the last node. We need to keep the address of the second last node also because this will become the new last node. For this purpose, a concept of the previous node is introduced here.



ALGORITHM DeleteEnd(START)

BEGIN:

p=START

q=NULL

WHILE p->Next!=NULL DO

q=p

p=p->Next

q->Next=NULL

x=p->Info

FreeNode(p)

RETURN x

END;

Time Complexity: $\theta(N)$

The while loop executes two statements repetitively until the last element is reached. Also, five more statements are executed outside the loop

Space Complexity: $\theta(1)$

Three extra variables namely p, q & x are utilized

- **Deletion of a node form the Mid:** It is assumed that the address of the node before the node to be Deleted is given. Pointer adjustments are required for maintaining the breakage of the link.

ALGORITHM DeleteAft(p)

BEGIN:

IF $p == \text{NULL}$ OR $p \rightarrow \text{Next} == \text{NULL}$ THEN

WRITE ("Void Deletion")

EXIT(1)

$q = p \rightarrow \text{Next}$

$x = q \rightarrow \text{Info}$

$p \rightarrow \text{Next} = q \rightarrow \text{Next}$

FreeNode(q)

RETURN x

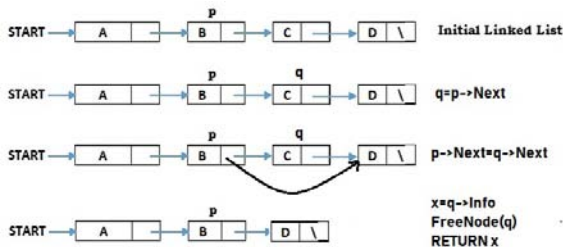
END;

Time Complexity: $\theta(1)$

Here, statements are executed

Space Complexity: $\theta(1)$

Two extra variables q & x are used apart from the parameters passed



- **Traversal:** It simply requires the visit of each node starting from the first node.

ALGORITHM Traverse(START)

BEGIN:

$p = \text{START}$

WHILE $p \neq \text{NULL}$ D

WRITE $p \rightarrow \text{Info}$

$p = p \rightarrow \text{Next}$

END;

Time Complexity: $\theta(N)$

It takes $2n$ statement executions by the while loop where n elements are present in the list. Also, one more statement is executed to initialize p

Space Complexity: $\theta(1)$

p is taken as an extra variable

C- Program

```
/*  
#include<stdio.h>  
#include<stdlib.h>  
*/  
struct node  
{  
int info;  
struct node *next;  
};  
/*  
struct node *GetNode()  
{  
struct node *p;  
p=(struct node *)malloc(sizeof(struct node));  
return p;  
}  
*/  
void InsertBeginning(struct node **START, int key)  
{  
struct node *p, *temporary;  
p=GetNode();  
p->info=key;  
p->next=(*START);  
(*START)=p;  
}
```

```

}
/*****/
VoidInsertEnd(struct node **START, int key)
{
    struct node *p, *q;
    p=*START;
    while(p→next!=NULL)
    {
        p=p→next;
    }
    q=GetNode();
    q→info=key;
    q→next=NULL;
    p→next=q;
}
/*****/
VoidInsertAfter(struct node **q, int x)
{
    struct node *p;
    p=GetNode();
    p→info=x;
    p→next=(*q)→next;
    (*q)→next=p;
}
/*****/
void DeleteBeginning(struct node **START)
{
    if((*START) == NULL)
    {
        printf("void DELETION\n");
        exit(1);
    }
    else
    {

```

```
struct node *p;
p=(*START);
(*START)=p→next;
int x=p→info;
free(p);
printf("Deleted element %d\n", x);
}
}
/*****/

void DeleteAfter(struct node **p)
{
if((*p) == NULL||(*p)→next == NULL)
{
printf("void DELETION\n");
exit(1);
}
else
{
struct node *q;
q=(*p)→next;
(*p)→next=q→next;
int x=(q)→info;
free(q);
printf("Deleted element %d\n", x);
}
}
/*****/

void DeleteEnd(struct node **START)
{
if((*START) == NULL)
{
printf("void DELETION\n");
exit(1);
}
```



```

else
{
struct node *p, *q;
p=(*START);
q=NULL;
while(p→next!=NULL)
{
q=p;
p=p→next;
}
if(q == NULL)
{
(*START)=NULL;
}
else
{
q→next=NULL;
}
int x=p→info;
free(p);
printf("Deleted element %d\n", x);
}
}

/*****/

void Traverse(struct node *START)
{
struct node *p;
p=(START);
while(p!=NULL)
{
printf("%d→", p→info);
p=p→next;
}
printf("NULL\n");

```

```
}  
int main()  
{  
    struct node *START;  
    struct node *p;  
    START=NULL;  
    InsertBeginning(&START, 700);  
    InsertBeginning(&START, 600);  
    InsertBeginning(&START, 100);  
    InsertBeginning(&START, 400);  
    p=START;  
    InsertBeginning(&START, 300);  
    InsertBeginning(&START, 100);  
    InsertBeginning(&START, 100);  
    InsertBeginning(&START, 50);  
    InsertEnd(&START, 750);  
    InsertAfter(&p, 350);  
    Traverse(START);  
  
    DeleteBeginning(&START);  
    Traverse(START);  
    DeleteEnd(&START);  
    Traverse(START);  
    DeleteAfter(&p);  
    Traverse(START);  
}
```

13.3. RELATED FUNCTIONS

- Pair-wise swapping of elements

ALGORITHM PairWiseSwap(START)

BEGIN:

p=START

WHILE p!=NULL && p→Next!=NULL DO

```

        Swap(p→ Info, p→ Next→ Info)
        p=p→ Next
    IF (p!=NULL) THEN
        p=p→ Next
    ELSE
        BREAK
    Traverse(START)
END;

```

Time Complexity: $\Theta(N)$

The while loop executes four statements in each iteration. It iterates from the first node of the linked list to the second last node. Also, the Traverse function has $\Theta(N)$ time complexity. Hence, the number of executed statements are $4(N-1) + CN$

Space Complexity: $\Theta(1)$

p is taken as an extra variable

- Printing the Contents of Linked List in Reverse Order

ALGORITHM RevTraverse(START)

BEGIN:

IF START!=NULL THEN

RevTraverse(START)

WRITE(Info(START))

END;

Time Complexity: $\Theta(N)$

Since this is a recursive function, a call stack of size n is formed where each time one statement is executed.

Space Complexity: $\Theta(N)$

The call stack formed has a height N where N is the number of elements in the linked list

C-Program

```

/*****
#include<stdio.h>

```

```
#include<stdlib.h>

/*****/

struct node
{
    int info;
    struct node *next;
};

/*****/

struct node *GetNode()
{
    struct node *p;
    p=(struct node *)malloc(sizeof(struct node));
    return p;
}

/*****/

void InsertBeginning(struct node **START, int key)
{
    struct node *p, *temporary;
    p=GetNode();
    p→info=key;
    p→next=(*START);
    (*START)=p;
}

/*****/

VoidInsertAfter(struct node **q, int x)
{
    struct node *p;
    p=GetNode();
    p→info=x;
    p→next=(*q)→next;
    (*q)→next=p;
}
```

```
void InsertAtPosition(struct node **START, int pos, int item)
```

```
{
    struct node *p;
    p=GetNode();
    p→info=item;
    if(*START == NULL)
    {
        *START=p;
    }
    else
    {
        int i=0;
        struct node *temporary=*START;
        while(temporary→next!=NULL)
        {
            i=i+1;
            if(i == pos-1)
            {
                p→next=temporary→next;
                temporary→next=p;
                break;
            }
            temporary=temporary→next;
        }
    }
}

/*****/
```

```
void DisplayReverse(struct node *START)
```

```
{

    if(START!=NULL)
    {
        DisplayReverse((START)→next);
    }
}
```

```
printf("%d→", (START)→info);
}

}

/*****/

void Traverse(struct node *START)
{
    struct node *p;
    p=(START);
    while(p!=NULL)
    {
        printf("%d→", p→info);
        p=p→next;
    }
    printf("NULL\n");
}

/*****/

int main()
{
    struct node *START;
    struct node *p;
    START=NULL;
    InsertBeginning(&START, 700);
    InsertBeginning(&START, 650);
    InsertBeginning(&START, 500);
    InsertBeginning(&START, 450);
    InsertBeginning(&START, 330);
    InsertBeginning(&START, 200);
    InsertBeginning(&START, 100);
    InsertBeginning(&START, 50);
    InsertAtPosition(&START, 2, 150);
    printf("start\n");

    Traverse(START);
```

```

struct node *START1=NULL, *START2=NULL;
InsertBeginning(&START1, 750);
InsertBeginning(&START1, 650);
InsertBeginning(&START1, 550);
InsertBeginning(&START1, 450);
InsertBeginning(&START1, 330);
printf("\nstart1\n");
Traverse(START1);

```

```

printf("\nreverse of start\n");
DisplayReverse(START);
printf("\nreverse of start1\n");
displayreverse(START1);
}

```

- Reversing the Linear Linked List

ALGORITHM ReverseLinkedList (START)

BEGIN:

current=START

previous=NULL

WHILE current!=NULL DO

 Next=current→Next

current→Next =previous

 previous=current

 current=Next

START=previous

RETURN START

END;

Time Complexity: $\Theta(N)$

The while loop executes four statements in each iteration. The loop runs N times where N is the number of elements in the linked list. Apart from the loop, four statements are executed in the algorithm

Space Complexity: $\theta(1)$

Only two extra variables namely current & previous are used

C-Program:-

```
/******  
#include<stdio.h>  
#include<stdlib.h>  
/******  
struct node  
{  
int info;  
struct node *next;  
};  
/******  
struct node *GetNode()  
{  
struct node *p;  
p=(struct node*)malloc(sizeof(struct node));  
return p;  
}  
/******  
void InsertBeg(struct node **START, int x)  
{  
struct node *temporary;  
temporary=GetNode();  
temporary->info=x;  
temporary->next=(*START);  
(*START)=temporary;  
}  
/******  
void Traverse(struct node *START)  
{  
struct node *t;  
t=START;
```



```

while(t!=NULL)
{
printf("%d\t", t→info);
t=t→next;
}
}
/*****/

struct node *reverse(struct node*START1)
{
struct node *curr, *prev, *next;
curr=START1;
prev=NULL;
while(curr!=NULL)
{
next=curr→next;
curr→next=prev;
prev=curr;
curr=next;
}
START1=prev;
return START1;
}
/*****/

int main()
{
struct node *START1, *START2;
START1=NULL;
START2=NULL;
printf("Linked list Before Reverse is...\n ");
InsertBeg(&START1, 10);
InsertBeg(&START1, 20);
InsertBeg(&START1, 30);
InsertBeg(&START1, 40);
InsertBeg(&START1, 50);

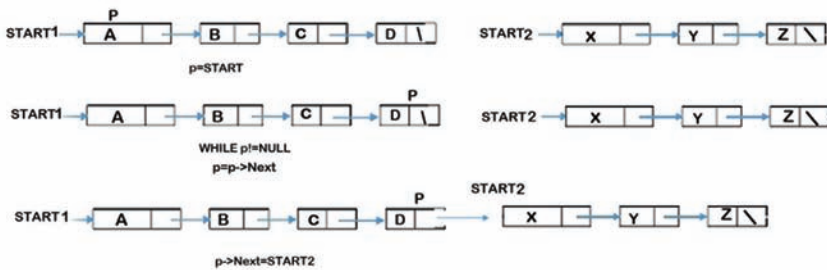
```

```

InsertBeg(&START1, 60);
Traverse(START1);
printf("\nLinked List After reverse is...\n");
START2 = reverse(START1);
Traverse(START2);
}
/*****/

```

- **Concatenation of Linear Linked List:** Concatenation of linear linked list requires the Traversal of the first Linked list to reach to the last node. The last node of the first linked list is linked with the first node of the second linked list then.



ALGORITHM ConcatList(list1, list2)

BEGIN:

p=list1

WHILE(p→ Next!=NULL) Do

p=p→ Next

p→ Next=list2

RETURN list1

END;

Time Complexity: $\Theta(N)$, N is the length of the first Linked List

In the above algorithm, the last element of the first list is connected to the first element of the second. For this, we reach the last element using a while loop, which runs N times. After that, two statements are executed

Space Complexity: $\Theta(1)$

An extra variable p is used

C-Program

```
/******  
#include<stdio.h>  
#include<stdlib.h>  
/******  
  
struct node  
{  
int info;  
struct node *next;  
};  
/******  
  
struct node *GetNode()  
{  
struct node *p;  
p=(struct node*)malloc(sizeof(struct node));  
return p;  
}  
/******  
  
void InsertBeg(struct node **START, int x)  
{  
struct node *temporary;  
temporary=GetNode();  
temporary→info=x;  
temporary→next=(*START);  
(*START)=temporary;  
}  
/******  
  
void Traverse(struct node *START)  
{  
struct node *t;  
t=START;  
while(t!=NULL)  
{  
printf(“%d\t”, t→info);
```

```
t=t→next;
}
}

/*****/
struct node *concat(struct node*list1, struct node*list2)
{
    struct node *t;
    t=list1;
    while(t→next!=NULL)
    {
        t=t→next;
    }
    t→next=list2;
    return list1;
}

/*****/
int main()
{
    struct node *START1, *START2, *START3;
    struct node *p;
    START1=NULL;
    START2=NULL;
    START3=NULL;
    printf("First Linked List is...\n");
    InsertBeg(&START1, 300);
    InsertBeg(&START1, 200);
    InsertBeg(&START1, 100);
    Traverse(START1);
    printf("\n");
    printf("Second Linked List is...\n");
    InsertBeg(&START2, 700);
    InsertBeg(&START2, 600);
    InsertBeg(&START2, 500);
    Traverse(START2);
```

```

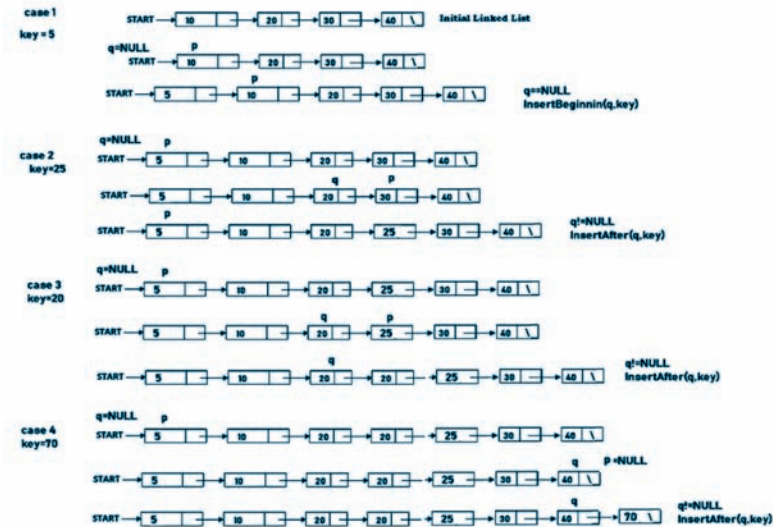
printf("\n");
printf("Concatination of two Linked List is...\n");
START3 = concat(START1, START2);
Traverse(START3);
}
/*****

```

- Creation of Ascending Order Linked List

Let us assume that an ascending order linked list is already there. For Insertion in such linked list, we need to search the appropriate position for insertion. There are the following possibilities:

- **If key to be inserted is less than the first node's data:** InsertBeginning function is called
- **Somewhere in between:** InsertAfter function is called
- **At the end:** InsertAfter function at the last node is called.



ALGORITHM AscendingOrderList (START, key)

BEGIN:

P=START

Q=NULL

WHILE P!=NULL AND key >= P → Info DO

```

        Q=P
        P=P→ Next
    IF Q == NULL THEN
        InsertBeginning(START, key)
    ELSE
        InsertAfter(Q, key)
END;

```

Time Complexity: $O(N)$, $\Omega(1)$

In the best case, the element to be inserted will be the smallest. Hence, the loop will not be entered. Only five statements will be executed.

In the worst case, the new element will be large from all the elements present in the list. Hence, the loop will run N times (where N is the number of nodes in the list). After that, two statements will be executed.

Space Complexity: $\theta(1)$

Number of extra variables used are two namely **p & q**

C- Program

```

/*****/
#include<stdio.h>
#include<malloc.h>
/*****/

struct node
{
    int info;
    struct node *next;
};
/*****/

struct node *GetNode()
{
    struct node *p;
    p=(struct node *)malloc(sizeof(struct node));
    return p;
}

```

```

}
/*****/

void InsertBeginning(struct node **START, int key)
{
    struct node *p, *temporary;
    p=GetNode();
    p→info=key;
    p→next=(*START);
    (*START)=p;
}
/*****/

void InsertAftervalue(struct node **START, int y, int x)
{
    struct node *q, *temporary;
    temporary=(*START);
    while(temporary!=NULL)
    {
        if(temporary→info == y)
        {
            break;
        }
        else
            temporary=temporary→next;
    }
    q=GetNode();
    q→info=x;
    q→next=(temporary)→next;
    (temporary)→next=q;
}
/*****/

void InsertAfter(struct node **q, int x)
{
    struct node *p;
    p=GetNode();

```

```
p→info=x;
p→next=(*q)→next;
(*q)→next=p;
}
/*****/
void OrderedInsert(struct node **START, int key)
{
    struct node *p, *q;
    p=(*START);
    q=NULL;
    while(p!=NULL&&key >= p→info)
    {
        q=p;
        p=p→next;
    }
    if(q == NULL)
        InsertBeginning(&(*START), key);
    else
        InsertAfter(&q, key);
}
/*****/
void Traverse(struct node *START)
{
    struct node *p;
    p=(START);
    while(p!=NULL)
    {
        printf("%d→", p→info);
        p=p→next;
    }
    printf("NULL\n");
}
/*****/
```



```
int main()
{
    struct node *START;
    struct node *p;
    START=NULL;
    InsertBeginning(&START, 700);
    InsertBeginning(&START, 600);
    InsertBeginning(&START, 500);
    InsertBeginning(&START, 400);
    InsertBeginning(&START, 300);
    InsertBeginning(&START, 200);
    InsertBeginning(&START, 100);
    Traverse(START);

    InsertAftervalue(&START, 300, 350);
    Traverse(START);

    OrderedInsert(&START, 150);
    Traverse(START);

    OrderedInsert(&START, 750);
    Traverse(START);

    OrderedInsert(&START, 650);
    Traverse(START);

    OrderedInsert(&START, 50);
    Traverse(START);
}
```

Merging Two Ascending Order Linked List: This process is much similar to the Merge Array operation discussed in Sorting Chapter. In this, we compare the elements of both the linked lists. Whichever is smaller is added to the third linked list in the end. In case comparisons are not possible, whichever linked list has

remaining nodes, its elements are added one by one onto the third linked list in the end.

ALGORITHM Merge(list1, list2)

BEGIN:

p=list1

q=list2

List 3=NULL

WHILE p!=NULL && q!=NULL DO

 IF(p→ Info<q→ Info) THEN

 InsertEnd(list3, p→ Info)

 p=p→ Next

 ELSE

 InsertEnd(list3, q→ Info)

 q=q→ Next

WHILE p!=NULL DO

 InsertEnd(list3, p→ Info)

 p=p→ Next

WHILE q!=NULL DO

 InsertEnd(list3, q→ Info)

 q=q→ Next

RETURN list3

END;

Time Complexity: $\Theta(M+N)$, M, and N is the length of first and Second linked list resp.

As we can see, to copy all the elements of both the linked lists, we need to iterate from each in every element. Hence, in any case, the loops together will make $m+n$ iterations.

Space Complexity: $\Theta(M+N)$

A new linked list of length $m+n$ is formed to accommodate all the elements of both the lists

InsertEnd here is rewritten as:

ALGORITHM InsertEnd(START, key)**BEGIN:**

p=START

IF START == NULL THEN

InsertBeginning(START, key)

ELSE

WHILE p!=NULL DO

p=p→ Next

q = GetNode()

q→Info=key

q→ Next=NULL

p→ Next=q

END;**C-Program**

/*****/

#include<stdio.h>

#include<malloc.h>

/*****/

struct node

{

int info;

struct node *next;

};

/*****/

struct node *GetNode()

{

struct node *p;

p=(struct node *)malloc(sizeof(struct node));

return p;

}

/*****/

```
void InsertBeginning(struct node **START, int key)
{
    struct node *p, *temporary;
    p=GetNode();
    p→info=key;
    p→next=(*START);
    (*START)=p;
}
/*****/

void InsertAftervalue(struct node **START, int y, int x)
{
    struct node *q, *temporary;
    temporary=(*START);
    while(temporary!=NULL)
    {
        if(temporary→info == y)
        {
            break;
        }
        else
            temporary=temporary→next;
    }
    q=GetNode();
    q→info=x;
    q→next=(temporary)→next;
    (temporary)→next=q;
}
/*****/

void InsertAfter(struct node **q, int x)
{
    struct node *p;
    p=GetNode();
    p→info=x;
    p→next=(*q)→next;
```

```

(*q)→next=p;
}
/*****/

struct node *mergelist(struct node **START, struct node **START1)
{
    struct node *p, *q, *d, *k;
    p>(*START);
    q>(*START1);
    d=NULL;
    if(p→info<q→info)
    {
        InsertBeginning(&d, p→info);
        p=p→next;
    }
    else
    {
        InsertBeginning(&d, q→info);
        q=q→next;
    }
    k=d;
    while(p!=NULL&&q!=NULL)
    {
        if(p→info>q→info)
        {
            InsertAfter(&d, q→info);
            d=d→next;
            q=q→next;
        }
        else
        {
            InsertAfter(&d, p→info);
            d=d→next;
            p=p→next;
        }
    }
}

```

```
}
while(p!=NULL)
{
InsertAfter(&d, p→info);
d=d→next;
p=p→next;
}
while(q!=NULL)
{
InsertAfter(&d, q→info);
d=d→next;
q=q→next;
}
return k;
}
/*****/
void Traverse(struct node *START)
{
struct node *p;
p=(START);
while(p!=NULL)
{
printf("%d→", p→info);
p=p→next;
}
printf("NULL\n");
}
/*****/
int main()
{
struct node *START;
struct node *p;
START=NULL;
InsertBeginning(&START, 700);
```

```

InsertBeginning(&START, 600);
InsertBeginning(&START, 500);
InsertBeginning(&START, 400);
InsertBeginning(&START, 300);
InsertBeginning(&START, 200);
InsertBeginning(&START, 100);
Traverse(START);

```

```

InsertAftervalue(&START, 300, 350);
Traverse(START);

```

```

struct node *START1=NULL, *START2=NULL;
InsertBeginning(&START1, 750);
InsertBeginning(&START1, 650);
InsertBeginning(&START1, 550);
InsertBeginning(&START1, 450);
InsertBeginning(&START1, 330);
Traverse(START1);

```

```

START2=mergelist(&START, &START1);
printf("after merge\n");
Traverse(START2);
}

```

13.4. SET OPERATIONS USING LINKED LIST

- Union of Two Ordered Linked Lists (Linked List Elements are Considered as Set)

ALGORITHM Union(START1, START2)

BEGIN:

START3=NULL

p=START

q=START2

WHILE p!=NULL&&q!=NULL DO

```

    IF p→ Info == q→ Info THEN
        p=p→ Next
        q=q→ Next
    ELSE
        IF p→ Info < q→ Info THEN
            InsertEnd(START3, p→ Info)
            p=p→ Next
        ELSE
            InsertEnd(START3, p→ Info)
            q=q→ Next
    WHILE p!=NULL DO
        InsertEnd(START3, p→ Info)
        p=p→ Next
    WHILE q!=NULL DO
        InsertEnd(START3, q→ Info)
        q=q→ Next
    RETURN list3
END;
```

Time Complexity: $\Theta(M+N)$, M , and N is the Length of first and Second linked list resp.

As we can see, to compare all the elements of both the linked lists, we need to iterate from each in every element. Hence, in any case, the loops together will make $m+n$ iterations.

Space Complexity: $\Theta(M+N)$

A new linked list of length $m+n$ is formed to accommodate the elements of both the lists

C- Program

```

/*****
#include<stdio.h>
#include<stdlib.h>
*****/
```



```

struct node
{
int info;
struct node *next;
};
/*****/

struct node *GetNode()
{
struct node *p;
p=(struct node *)malloc(sizeof(struct node));
return p;
}
/*****/

void InsertBeginning(struct node **START, int key)
{
struct node *p, *temporary;
p=GetNode();
p→info=key;
p→next=(*START);
(*START)=p;
}
/*****/

void InsertAfter(struct node **q, int x)
{
struct node *p;
p=GetNode();
p→info=x;
p→next=(*q)→next;
(*q)→next=p;
}
/*****/

struct node *union1(struct node **START, struct node **START1)
{
struct node *p, *q, *d, *k;

```

```
p=(*START);
q=(*START1);
d=NULL;
if(p→info == q→info)
{
    InsertBeginning(&d, q→info);
    q=q→next;
}
else if(p→info<q→info)
{
    InsertBeginning(&d, p→info);
    p=p→next;
}
else
{
    InsertBeginning(&d, q→info);
    q=q→next;
}
k=d;
while(p!=NULL&&q!=NULL)
{
    if(p→info>q→info)
    {
        InsertAfter(&d, q→info);
        d=d→next;
        q=q→next;
    }
    else if(p→info<q→info)
    {
        InsertAfter(&d, p→info);
        d=d→next;
        p=p→next;
    }
    else if(p→info == q→info)
```

```

{
InsertAfter(&d, p→info);
d=d→next;
q=q→next;
p=p→next;
}
}
while(p!=NULL)
{
InsertAfter(&d, p→info);
d=d→next;
p=p→next;
}
while(q!=NULL)
{
InsertAfter(&d, q→info);
d=d→next;
q=q→next;
}
return k;
}
/*****
void Traverse(struct node *START)
{
struct node *p;
p=(START);
while(p!=NULL)
{
printf("%d→", p→info);
p=p→next;
}
printf("NULL\n");
}
int main()

```

```
{
struct node *START;
struct node *p;
START=NULL;
InsertBeginning(&START, 700);
InsertBeginning(&START, 650);
InsertBeginning(&START, 500);
InsertBeginning(&START, 450);
InsertBeginning(&START, 330);
InsertBeginning(&START, 200);
InsertBeginning(&START, 100);
InsertBeginning(&START, 50);
Traverse(START);

struct node *START1=NULL, *START2=NULL;
InsertBeginning(&START1, 750);
InsertBeginning(&START1, 650);
InsertBeginning(&START1, 550);
InsertBeginning(&START1, 450);
InsertBeginning(&START1, 330);
Traverse(START1);

START2=union1(&START, &START1);
printf("after union\n");
Traverse(START2);
}
```

- Intersection of Two Ordered Linked List (Linked Lists are Considered as Set)

ALGORITHM Intersection(list1, list2)

BEGIN:

list3=NULL

p=list1

q=list2

```

WHILE p!=NULL && q!=NULL DO
    IF p→ Info == q→ Info THEN
        InsertEnd(list3, p→ Info)
        p=p→ Next
        q=q→ Next
    ELSE
        IF p→ Info<q→ Info THEN
            p=p→ Next
        ELSE
            q=q→ Next
RETURN list3
END:

```

Time Complexity: $\Theta(M+N)$, M, and N is the Length of first and Second linked list resp.

As we can see, to compare all the elements of both the linked lists, we need to iterate from each in every element. Hence, in any case, the loops together will make $m+n$ iterations.

Space Complexity: $\Theta(M+N)$

A new linked list of length $m+n$ is formed to accommodate the elements of both the lists

C-Program

```

/*****/
#include<stdio.h>
#include<stdlib.h>
/*****/

struct node
{
    int info;
    struct node *next;
};
/*****/

```

```
struct node *GetNode()
{
    struct node *p;
    p=(struct node*)malloc(sizeof(struct node));
    return p;
}
/*****/

void InsertBeg(struct node **START, int x)
{
    struct node *temporary;
    temporary=GetNode();
    temporary→info=x;
    temporary→next=(*START);
    (*START)=temporary;
}
/*****/

void Traverse(struct node *START)
{
    struct node *t;
    t=START;
    while(t!=NULL)
    {
        printf("%d\t", t→info);
        t=t→next;
    }
}
/*****/

struct node *Intersection(struct node*list1, struct node*list2)
{
    struct node *list3=NULL;
    struct node *p, *q;
    p=list1;
    q=list2;
    while(p!=NULL&&q!=NULL)
```

```

{
if(p→info == q→info)
{
InsertBeg(&list3, p→info);
p=p→next;
q=q→next;
}
else
{
if(p→info<q→info)
{
p=p→next;
}
else
{
q=q→next;
}
}
}
return list3;
}
/*****/

int main()
{
struct node *START1, *START2, *START3, *t;
START1=NULL;
START2=NULL;
START3=NULL;
printf("First Linked is...\n");
InsertBeg(&START1, 10);
InsertBeg(&START1, 20);
InsertBeg(&START1, 30);
InsertBeg(&START1, 40);
InsertBeg(&START1, 50);

```

```

InsertBeg(&START1, 60);
InsertBeg(&START2, 12);
InsertBeg(&START2, 22);
InsertBeg(&START2, 40);
InsertBeg(&START2, 50);
InsertBeg(&START2, 60);
Traverse(START1);

printf("\nSecond Linked List is...\n");
Traverse(START2);
printf("\nIntersection Of Linked List is...\n");
START3=Intersection(START1, START2);
Traverse(START3);
}
/*****/

```

13.5. POLYNOMIAL REPRESENTATION AND POLYNOMIAL ADDITION

Polynomials can be represented using the Linked List. For this purpose, the linked list nodes are supposed to contain three fields

- Coefficient;
- Exponent; and
- Next address field.

Coefficient	Exponent	Next
Coef	Exp	Next

Examples of the degree 1 polynomials and their representation in the form of linked list given below.



For addition of the polynomials, if the exponents of the two polynomial terms are same, their coefficients are added. Usually, polynomials are written in decreasing order of their exponents.

ALGORITHM PolynomialAddition(Poly1, Poly2)**BEGIN:**

p=Poly1

q=Poly2

Poly3=NULL

WHILE p!=NULL AND q!=NULL DO

IF p→Exp == q→Exp THEN

InsertEnd(Poly3, p→coef+q→coef, p→Exp)

p=p→Next

q=q→Next

ELSE

IF p→Exp>q→Exp THEN

InsertEnd(Poly3, p→coef, p→Exp)

p=p→Next

ELSE

InsertEnd(Poly3, q→coef, q→Exp)

q=q→Next

WHILE p!=NULL DO

InsertEnd(Poly3, p→coef, p→Exp)

p=p→Next

WHILE q!=NULL DO

InsertEnd(Poly3, q→coef, q→Exp)

q=q→Next

RETURN Poly3

END;

The above Algorithm works more like the Merging of Sorted Arrays. The Algorithm can be improved by changing the InsertEnd by InsertAft with some changes in the start of the Algorithm. InsertEnd function will increase complexity as it has to Traverse the list every time it is called.

Another solution may be of Adding Elements in the Poly3 at the Beginning and

before returning, reverse the answer Polynomial.

Time Complexity $\Theta(m+n)$

If First Polynomial contains m elements and second one n , then one node will be added after one comparison. With each comparison either InsertAft or InsertBeginning has to be called (both require constant time) and some Increment in p or q to point to the next node (constant time operation). Total statements to be executed will be some order of $(m+n)$.

Space Complexity $\Theta(m+n)$

A third Polynomial whose size will be $m+n$ and variables p , q are required. Total space $m+n+2$

C-Program

```

/*****
#include<stdio.h>
#include<stdlib.h>
*****/
struct node
{
    int coef;
    int exp;
    struct node *next;
};
/*****
struct node * GetNode()
{
    struct node *p;
    p=(struct node *)malloc(sizeof(struct node));
    return p;
}
*****/
InsBeg(struct node **list, int c, int e)
{
    struct node *temporary;

```

```

temporary=GetNode();
temporary→coef=c;
temporary→exp=e;
temporary→next=*list;
*list=temporary;
}
/*****/
InsertEnd(struct node **list, int c, int e)
{
    struct node *temporary, *p;
    temporary=*list;
    if(*list == NULL)
        InsBeg(&(*list), c, e);
    else
    {
        while(temporary→next!=NULL)
            temporary=temporary→next;

        p=GetNode();
        p→coef=c;
        p→exp=e;
        p→next=NULL;
        temporary→next=p;
    }

}
/*****/
Traverse(struct node *list)
{
    struct node *t;
    t=list;
    while(t!=NULL)
    {
        printf("\t %dX%d +", t→coef, t→exp);
    }
}

```

```
        t=t→next;
    }
}
/*****/
struct node * AddPolynomial(struct node *poly1, struct node *poly2)
{
    struct node *poly3=NULL;
    struct node *p, *q;
    p=poly1;
    q=poly2;

    while(p!=NULL&&q!=NULL)
    {
        if(p→exp == q→exp)
        {
            InsertEnd(&poly3, p→coef+q→coef, p→exp);
            p=p→next;
            q=q→next;
        }
        else
        {
            if(p→exp>q→exp)
            {
                InsertEnd(&poly3, p→coef, p→exp);
                p=p→next;
            }
            else
            {
                InsertEnd(&poly3, q→coef, q→exp);
                q=q→next;
            }
        }
    }
}
```

```

        while(p!=NULL)
        {
InsertEnd(&poly3, p->coef, p->exp);
        p=p->next;
        }

        while(q!=NULL)
        {
InsertEnd(&poly3, q->coef, q->exp);
        q=q->next;
        }
        return poly3;
}
/*****/
void main()
{
    struct node *Start1, *Start2, *Start3;
    Start1=NULL;
    Start2=NULL;
    Start3=NULL;
    int x;

    InsertEnd(&Start1, 3, 8);
    InsertEnd(&Start1, 5, 7);
    InsertEnd(&Start1, -2, 6);
    InsertEnd(&Start1, 8, 4);
    printf("\nFirst Polynomial is:=> ");
    Traverse(Start1);

    InsertEnd(&Start2, 4, 7);
    InsertEnd(&Start2, 5, 6);
    InsertEnd(&Start2, -1, 3);
    InsertEnd(&Start2, 7, 2);

```

```

    InsertEnd(&Start2, -3, 0);
    printf("\n\n");
    printf("Second Polynomial is:=> ");
    Traverse(Start2);

    Start3=AddPolynomial(Start1, Start2);
    printf("\n\n");
    printf("Result of Polynomial Addition is:=> ");
    Traverse(Start3);
}
/*****/

```

13.6. LONG NUMBERS ARITHMETIC

Consider a very large number, e.g., 12345678912345678912345678912345. This number cannot be stored in any variable of variant integer. If two such numbers are to be added, storage of such a big number in the Linked List will make the arithmetic easy.

Consider a 2 byte integer in C which can have a range -2^{15} to $2^{15}-1$, i.e., -32768 to 32767 . That means the largest number of 5 digits cannot be stored in an integer variable. Divide the given number by dividing it into parts of 4 digit each from the back side. Store the number in the linked list. e.g.,

1234 5678 9123 4567 8912 7891 2345

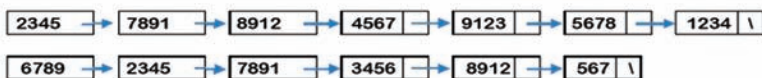


Addition of the numbers are performed from LSDs. Here the groups of 4 digits are stored from the back of the numbers. Hence addition will be performed by moving in the linked list from FRONT.

Suppose the Addition of Two numbers

1234 5678 9123 4567 8912 7891 2345

567 8912 3456 7891 2345 6789



For Addition: If two four-digit numbers added with an initial carry 0, it could generate a maximum of five-digit number. We can segregate last four digits are sum

and 1st digit is the carry for next additions.

$$8912+7891+0=16803$$

Sum is 6803 and 1 is carry for next addition

$$\text{Total}=\text{Number1}+\text{Number2}+\text{Carry}$$

$$\text{Sum}=\text{Total}\%10000$$

$$\text{Carry}=\text{Total}/10000$$

If largest numbers of 4 digits are added with initial carry 1

$$\text{Total}=9999+9999+1$$

$$= 19999$$

The Total remains in the range of integer variable.

$$\text{Sum}=9999$$

$$\text{Carry}=1$$

ALGORITHM AdditionOfLongNumbers(List1, List2)

BEGIN:

$$P=\text{List1}$$

$$Q=\text{List2}$$

$$\text{Carry}=0$$

$$\text{List3}=\text{NULL}$$

WHILE $p \neq \text{NULL}$ AND $q \neq \text{NULL}$ DO

$$\text{Total}=p \rightarrow \text{Info} + q \rightarrow \text{Info} + \text{Carry}$$

$$\text{Sum}=\text{Total}\%10000$$

$$\text{Carry}=\text{Total}/10000$$

$$\text{InsertBeginning}(\text{List3}, \text{Sum})$$

$$P=p \rightarrow \text{Next}$$

$$Q=q \rightarrow \text{Next}$$

WHILE $p \neq \text{NULL}$ DO

$$\text{Total}=p \rightarrow \text{Info} + \text{Carry}$$

$$\text{Sum}=\text{Total}\%10000$$

$$\text{Carry}=\text{Total}/10000$$

$$\text{InsertBeginning}(\text{List3}, \text{Sum})$$

$$P=p \rightarrow \text{Next}$$

```
WHILE q!=NULL DO
    Total=q→Info + Carry
Sum=Total%10000
Carry=Total/10000
InsertBeginning(List3, Sum)
q=q→Next

IF Carry == 1
    InsertBeginning(List3, Carry)

Reverse(List3)
RETURN List3
END;
```

13.7. LINKED LIST IMPLEMENTATION OF QUEUE

For Implementation of Queue as Linked List, we have two ends REAR and FRONT. Both keep the address of the Node. Insertions take place at the REAR end and Deletions take place at the FRONT end. Here Insertions are taking place after the REAR and deletions from the Beginning ensuring the FIFO behavior.

ALGORITHM Initialize (FRONT, REAR, x)

BEGIN:

REAR =NULL

FRONT =NULL

END;

Time Complexity: $\Theta(1)$

Only two statements are executed

Space Complexity: $\Theta(1)$

No extra variable is used

ALGORITHM Empty(FRONT, REAR, x)

BEGIN:

IF (REAR == NULL && FRONT == NULL) THEN


```
RETURN True
    ELSE
        RETURN False
END;
```

Time Complexity: $\theta(1)$

Two statements are executed in any case

Space Complexity: $\theta(1)$

No extra variable is used

ALGORITHM Enqueue(FRONT, REAR, x)**BEGIN:**

```
IF REAR == NULL THEN
```

```
    InsertBeginning(REAR, x)
```

```
    FRONT = REAR
```

```
ELSE
```

```
    InsertAfter(REAR, x)
```

```
    REAR = REAR → Next
```

```
END;
```

Time Complexity: $\theta(1)$

Three statements are executed in any case

Space Complexity: $\theta(1)$

The space is allocated in memory to the new node

ALGORITHM Dequeue(FRONT, REAR)**BEGIN:**

```
IF FRONT == NULL THEN
```

```
    WRITE("Queue Underflows")
```

```
    Exit(1)
```

```
ELSE
```

```
    x = DelBeg(FRONT)
```

```
IF FRONT == NULL
REAR = NULL
RETURN x
END;
```

Time Complexity: $\Theta(1)$

If the list is Empty, three statements are executed. Otherwise, 5 are executed.

Space Complexity: $\Theta(1)$

An extra variable **x** is used

C-Program

```
/*******/
#include<stdio.h>
#include<stdlib.h>
/*******/
struct node
{
int info;
struct node *next;
};
/*******/
struct node *GetNode()
{
struct node *p;
p=(struct node*)malloc(sizeof(struct node));
return p;
}
/*******/
void InsertBeg(struct node **START, int x)
{
struct node *temporary;
temporary=GetNode();
temporary->info=x;
```

```

temporary→next = (*START);
(*START) = temporary;
}
/*****/
int DelBeg(struct node**START)
{
int x;
struct node*temporary;
temporary = (*START);
(*START) = (*START)→next;
x=temporary→info;
free(temporary);
return x;
}
/*****/
void Traverse(struct node *START)
{
struct node *t;
t=START;
while(t!=NULL)
{
printf("%d\t", t→info);
t = t→next;
}
printf("\n");
}
/*****/
Void InsertAfter(struct node **p, int key)
{
struct node *q;
q=GetNode();
q→info=key;
q→next=(*p)→next;
(*p)→next=q;
}

```

```
}
/*****/
void Enqueue(struct node **FRONT, struct node **REAR, int x)
{
    if(*REAR == NULL)
    {
        InsertBeg(&(*REAR), x);
        *FRONT=*REAR;
    }
    else
    {
        instaf(&(*REAR), x);
        (*REAR)=(*REAR)→next;
    }
}
/*****/
int Empty(struct node **FRONT, struct node **REAR)
{
    if((*REAR) == NULL&&(*FRONT) == NULL)
        return 1;
    else
        return 0;
}
/*****/
int Dequeue(struct node**FRONT, struct node**REAR)
{
    int x;
    if(*FRONT == NULL)
    {
        printf("Deletion is not possible");
        exit(1);
    }
    else
    {
```

```

x=DelBeg(&(*FRONT));
if(*FRONT == NULL)
*REAR=NULL;
}
return x;
}
/*****/
int main()
{
struct node*FRONT, *REAR;
FRONT=NULL; //INITIALIZATION
REAR=NULL;
int x;
Enqueue(&FRONT, &REAR, 100);
Enqueue(&FRONT, &REAR, 200);
Enqueue(&FRONT, &REAR, 300);
Enqueue(&FRONT, &REAR, 400);
Enqueue(&FRONT, &REAR, 500);
Traverse(FRONT);

x=Dequeue(&FRONT, &REAR);
printf("\nDeleted value is...%d\n", x);
Traverse(FRONT);
Enqueue(&FRONT, &REAR, 700);
Traverse(FRONT);

x=Dequeue(&FRONT, &REAR);
printf("\nDeleted value is...%d\n", x);
Traverse(FRONT);
}

```

13.8. LINKED LIST IMPLEMENTATION OF STACK

For Implementation of Stack as Linked List, we have one ends TOP. It keeps the

address of the Node. Insertions and Deletions both take place at the TOP end. Here Insertions are taking place at the beginning and deletions at the beginning as well ensuring the LIFO behavior.

ALGORITHM Initialize()**BEGIN:**

TOP=NULL

END;**Time Complexity: $\Theta(1)$**

One statement is executed

Space Complexity: $\Theta(1)$

No new variable is used.

ALGORITHM Empty()**BEGIN:**

IF Top == NULL THEN

RETURN TRUE

ELSE

RETURN FALSE

END;**Time Complexity: $\Theta(1)$**

Two statements are executed

Space Complexity: $\Theta(1)$

No new variable is used

ALGORITHM Push(Top, x)**BEGIN:**

InsertBeginning(Top, x)

END;

Time Complexity: $\Theta(1)$

A function is called whose time complexity is $T(1)$

Space Complexity: $\Theta(1)$

No new variable is used. Also, the space is allocated in the memory to the new node in InsertBeginning Function.

ALGORITHM Pop(Top)**BEGIN:**

IF Top == NULL THEN

 WRITE("Stack Underflows")

 Exit(1)

ELSE

 x=DelBeg(Top)

RETURN x

END;

ALGORITHM StackTop(Top)**BEGIN:**

RETURN(Info(Top))

END;

Time Complexity: $\Theta(1)$

Only one statement is executed, i.e., **RETURN** statement

Space Complexity: $\Theta(1)$

No new variable is used

C-Program

```
/******
```

```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
/******
```

```
struct node
```

```
{
int info;
struct node *next;
};
/*****/

struct node *GetNode()
{
struct node *p;
p=(struct node*)malloc(sizeof(struct node));
return p;
}
/*****/

void InsertBeg(struct node **START, int x)
{
struct node *temporary;
temporary=GetNode();
temporary→info=x;
temporary→next=(*START);
(*START)=temporary;
}
/*****/

int DelBeg(struct node**START)
{
int x;
struct node*temporary;
temporary=(*START);
(*START)=(*START)→next;
x=temporary→info;
free(temporary);
return x;
}
/*****/

void Traverse(struct node *START)
{
```



```

struct node *t;
t=START;
while(t!=NULL)
{
printf("%d\t", t→info);
t=t→next;
}
}
/*****/

void Push(struct node **TOP, int x)
{
InsertBeg(&(*TOP), x);
}
/*****/

int StackTop(struct node *TOP)
{
return TOP→info;
}
/*****/

int Empty(struct node **TOP)
{
if(*TOP == NULL)
return 1;
else
return 0;
}
/*****/

int Pop(struct node **TOP)
{
if(*TOP == NULL)
{
printf("\nStack Underflows\n");
exit(1);
}
}

```

```
else
{
    int x;
    x=DelBeg(&(*TOP));
    return x;
}
}
/*****/

int main()
{
    struct node *TOP;
    TOP=NULL;//INITIALIZATION
    int x;
    Push(&TOP, 50);
    Push(&TOP, 100);
    Push(&TOP, 200);
    Push(&TOP, 300);
    Traverse(TOP);
    //printf("\nDeleted value is..\n");

    x=Pop(&TOP);
    printf("\nDeleted value is..%d\n", x);
    Traverse(TOP);

    int y=Pop(&TOP);
    printf("\nDeleted value is..%d\n", y);
    Traverse(TOP);

    int z=StackTop(TOP);
    printf("\nTop value is..%d\n", z);
}
/*****/
```

13.9. LINKED LIST IMPLEMENTATION OF PRIORITY QUEUE

For the implementation of Priority Queue using linked list, consider an ascending order linked list. Insertions here are taking place at the appropriate place and Deletions from the beginning ensuring the service (Deletion) to the highest priority element at the moment

ALGORITHM PQInsert (PQ, key)

BEGIN:

P=PQ

Q=NULL

WHILE P!=NULL AND key >= P→ Info DO

Q=P

P=P→ Next

IF Q == NULL THEN

InsertBeginning(PQ, key)

ELSE

InsertAfter(Q, key)

END;

ALGORITHM PQDelete (PQ)

BEGIN:

x=DeleteBeginning(PQ)

RETURN x

END;

Time Complexity: O(1)

Statements in DeleteBeginning are constant in numbers

Space Complexity: $\theta(1)$

Number of extra variables used are 1 namely x

C- Program

/*******/

```
#include<stdio.h>
#include<malloc.h>
/*****/

struct node
{
    int info;
    struct node *next;
};
/*****/

struct node *GetNode()
{
    struct node *p;
    p=(struct node *)malloc(sizeof(struct node));
    return p;
}
/*****/

int PQDelete(struct node **PQ)
{
    int x=DeleteBeginning(&(*PQ));
    return x;
}
/*****/

int DeleteBeginning(struct node **START)
{
    if((*START) == NULL)
    {
        printf("void DELETION\n");
        exit(1);
    }
    else
    {
        struct node *p;
        p=(*START);
        (*START)=p->next;
```

```

int x=p→info;
free(p);
return x;
}
}
/*****/

void InsertBeginning(struct node **START, int key)
{
    struct node *p, *temporary;
    p=GetNode();
    p→info=key;
    p→next=(*START);
    (*START)=p;
}
/*****/

void InsertAftervalue(struct node **START, int y, int x)
{
    struct node *q, *temporary;
    temporary=(*START);
    while(temporary!=NULL)
    {
        if(temporary→info == y)
        {
            break;
        }
        else
            temporary=temporary→next;
    }
    q=GetNode();
    q→info=x;
    q→next=(temporary)→next;
    (temporary)→next=q;
}
/*****/

```

```
void InsertAfter(struct node **q, int x)
{
    struct node *p;
    p=GetNode();
    p→info=x;
    p→next=(*q)→next;
    (*q)→next=p;
}
/*****/

void PQInsert(struct node **PQ, int key)
{
    struct node *p, *q;
    p=(*PQ);
    q=NULL;
    while(p!=NULL&&key >= p→info)
    {
        q=p;
        p=p→next;
    }
    if(q == NULL)
        InsertBeginning(&(*PQ), key);
    else
        InsertAfter(&q, key);
}
/*****/

void Traverse(struct node *START)
{
    struct node *p;
    p=(START);
    while(p!=NULL)
    {
        printf("%d→", p→info);
        p=p→next;
    }
}
```

```

    printf("NULL\n");
}
/*****/

int main()
{
    struct node *PQ;
    PQ=NULL;
    PQInsert(&PQ, 150);
    Traverse(PQ);

    PQinsert(&PQ, 750);
    Traverse(PQ);

    PQInsert(&PQ, 650);
    Traverse(PQ);

    PQInsert(&PQ, 50);
    Traverse(PQ);

    x=PQDelete(&PQ);
    printf("Deleted Element from Priority Queue is => %d", x)
    Traverse(PQ);

}
/*****/

```

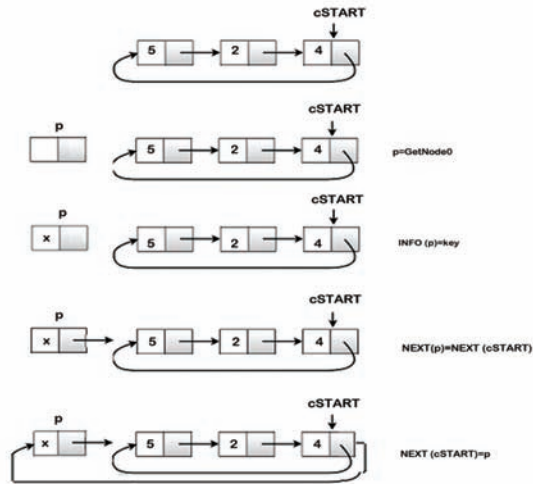
13.10. CIRCULAR LINKED LIST

13.10.1. Primitive Operations

In Circular Linked List, initialization is done with cSTART=NULL. cSTART remains at the last node in contrast to Linear List where START remains at First node. Keeping cSTART at the last node improves the run time of InsertEnd operation.

- **Insertion at the Beginning:** In case the List is empty initially, it is

made to point itself and cSTART is set at this node only



ALGORITHM InsertBeginning(cSTART, key)

BEGIN:

`p=GetNode()`

`p → Info=key`

IF `cSTART!=NULL` THEN

`p → Next=cSTART → Next`

`cSTART → Next=p`

ELSE

`cSTART=p`

`cSTART → Next=cSTART`

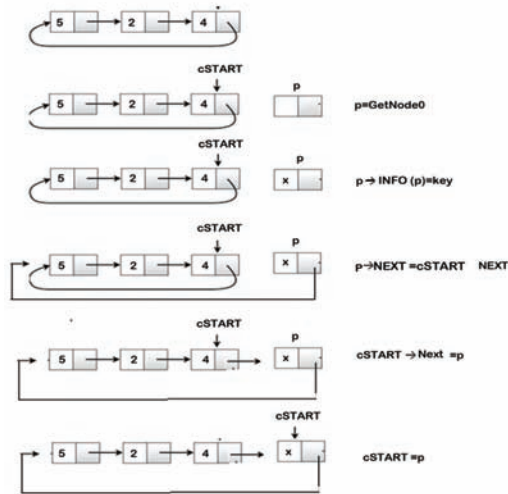
END;

Time Complexity: $\Theta(1)$

Here, in any case 4 statements are executed

Space Complexity: $\Theta(1)$

An extra variable **p** is used. Also, the space is allocated in the memory to the new node



- **Insertion at the END:** This simply requires shifting of $cSTART$ to the newly created node. In case the List is empty initially, it is made to point itself and $cSTART$ is set at this node only

ALGORITHM InsertEnd(cstart, key)

BEGIN:

$p = \text{GetNode}()$

$p \rightarrow \text{Info} = \text{key}$

IF $cSTART \neq \text{NULL}$ THEN

$p \rightarrow \text{Next} = cSTART \rightarrow \text{Next}$

$cSTART \rightarrow \text{Next} = p$

$cSTART = p$

ELSE

$cSTART = p$

$p \rightarrow \text{Next} = p$

END;

Time Complexity: $\Theta(1)$

If the list is Empty, 6 statements are executed. Else, 5 statements. Hence, they are not linearly dependent on the number of elements present in the list

Space Complexity: $\Theta(1)$

An extra variable p is used. Also, the space is allocated in the memory to the new

node

- **Insertion After a Given Node:** This operation works same as in Linear Linked List

ALGORITHM InsertAfter(cSTART, key)

BEGIN:

IF $p = \text{NULL}$ THEN

WRITE("void Insertion")

EXIT(1)

ELSE

GetNode()

$q \rightarrow \text{Info} = \text{key}$

$q \rightarrow \text{Next} = p \rightarrow \text{Next}$

$p \rightarrow \text{Next} = q$

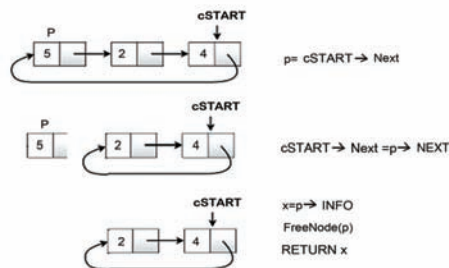
END;

Time Complexity: $\theta(1)$

If the list is Empty, 2 statements are executed. Else, 4 statements conditionally. Hence, they are not linearly dependent on the number of elements present in the list

Space Complexity: $\theta(1)$

Two extra variables p and q are utilized. Also, the space is allocated in the memory to the new node



a. Deletion Operations:

- **Deletion from the Beginning:** Next of $cSTART$ is made to point to second node. In case, list has only 1 node, $cSTART$ is set to NULL after this deletion

ALGORITHM Delbeg(cSTART)

BEGIN:

```
IF cSTART == NULL THEN
```

```
    EXIT(1)
```

```
ELSE
```

```
p=cSTART → Next
```

```
cSTART → Next=p → Next
```

```
    x=p → Info
```

```
IF cSTART → Next == cSTART THEN
```

```
    cSTART=NULL
```

```
FreeNode(p)
```

```
RETURN x
```

```
END;
```

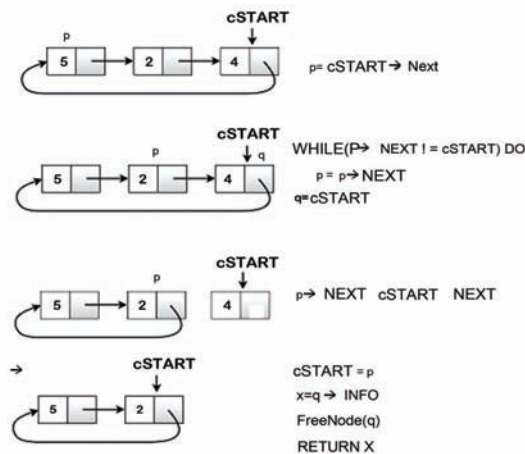
Time Complexity: $\Theta(1)$

It takes only two statements when the Circular linked list is Empty and 6 otherwise.

In both of the above cases, the number is fixed

Space Complexity: $\Theta(1)$

Two extra variables **p** & **x** are utilized



- **Deletion at the End:** Linked list is required to be traversed in order to reach the second last node. Next of second last node is made to point to the first node. **cSTART** now keeps the address of second last node.

ALGORITHM DeleteEnd(cSTART)**BEGIN:**

IF cSTART == NULL THEN

Write("void deletion")

EXIT(1)

ELSE

IF cSTART → Next == cSTART THEN

q=cSTART

cSTART=NULL

ELSE

P=cSTART → Next

WHILE P → Next != cSTART DO

P=P → Next

q=cSTART

P → Next=cSTART → Next

cSTART=p

x=q → Info

FreeNode(q)

RETURN x

END;**Time Complexity:** $\theta(N)$

The while loop is the dominating factor here which iterates N times running 3 statements in each iteration

Space Complexity: $\theta(1)$

Extra variables taken here are P, q & x

- **Deletion After a Given Node:** This operation works same as in Linear Linked List

ALGORITHM DelAft(cSTART)**BEGIN:**

p=cSTART

IF p == NULL || p → Next == NULL THEN

```
Write("void deletion")
EXIT(1)
ELSE
q=p→ Next
p→ Next=q→ Next
x=q→ Info
FreeNode(q)
RETURN x
END;
```

Time Complexity: $\Theta(1)$

Fixed number of statements are executed in any case which are linearly independent of the elements present in the linked list.

Space Complexity: $\Theta(1)$

Three extra variables **P**, **q** & **x** are used

- **Traversal:** These operations requires the printing of information of the nodes in a loop. node of last node is required to be printed separately

ALGORITHM Traverse(cSTART)**BEGIN:**

```
IF START!=NULL THEN
```

```
p=cSTART→ Next
```

```
WHILE p!=cSTART DO
```

```
    WRITE(p→ Info)
```

```
    p=p→ Next
```

```
WRITE(p→ Info)
```

```
END;
```

Time Complexity: $\Theta(N)$

Where N is the number of nodes in the list, The while loop iterates N times executing two statements in each iteration

Space Complexity: $\theta(1)$

p is taken as an extra variable

C-Program

```
/*  
#include<stdio.h>  
#include<stdlib.h>  
/*  
struct node  
{  
int info;  
struct node *next;  
};  
/*  
struct node *GetNode()  
{  
struct node *p;  
p=(struct node*)malloc(sizeof(struct node));  
return p;  
}  
/*  
void cInsertBeg(struct node **cSTART, int x)  
{  
struct node *temporary;  
temporary=GetNode();  
temporary->info=x;  
if(*cSTART == NULL)  
{  
(*cSTART)=temporary;  
(*cSTART)->next=(*cSTART);  
}  
else  
{
```

```

temporary→next=(*cSTART)→next;
(*cSTART)→next=temporary;
}
}

/*****/

void cInsertaft(struct node **cSTART, struct node **p, int x)
{
if((*cSTART) == NULL)
{
printf("void deletion\n");
exit(1);
}

else
{
struct node *temporary, *a;
temporary=GetNode();
temporary→info=x;
a=(*p)→next;
(*p)→next=temporary;
temporary→next=a;
}
}

/*****/

void cTraverse(struct node *cSTART)
{
struct node *t;
t=cSTART;
if(t!=NULL)
{
t=cSTART→next;
while(t!=cSTART)
{
printf("%d→", t→info);

```

```
t=t→next;
}
printf(“%d\n”, t→info);
}
}
/*****/
int cDeletebeg(struct node **cSTART)
{
if((*cSTART) == NULL)
{
printf(“void deletion\n”);
exit(1);
}
else
{
struct node *p;
p=(*cSTART)→next;
(*cSTART)→next=p→next;
int x=p→info;
if((*cSTART)→next == (*cSTART))
(*cSTART)=NULL;
free(p);
return x;
}
}
/*****/
int cDeleteEnd(struct node **cSTART)
{
if((*cSTART) == NULL)
{
printf(“void deletion\n”);
exit(1);
}
else
```



```

{
struct node *q;
if((*cSTART)→next == (*cSTART))
{
q=(*cSTART);
(*cSTART)=NULL;
free(q);
return(q→info);
}
else
{
struct node *p;
p=(*cSTART)→next;
while(p→next!=(*cSTART))
{
p=p→next;
}
q=(*cSTART);
p→next=(*cSTART)→next;
(*cSTART)=p;
}
int x=q→info;
free(q);
return x;
}
}
/*****/
void cInsertEnd(struct node **cSTART, int key)
{
struct node *temporary;
temporary=GetNode();
temporary→info=key;
if(*cSTART == NULL)
{

```

```

(*cSTART)=temporary;
(*cSTART)→next=(*cSTART);
}
else
{
temporary→next=(*cSTART)→next;
(*cSTART)→next=temporary;
(*cSTART)=temporary;
}
}
/*****/
int main()
{
struct node *cSTART1;
cSTART1=NULL;
cInsertBeg(&cSTART1, 300);
cInsertBeg(&cSTART1, 200);
cInsertBeg(&cSTART1, 100);
cTraverse(cSTART1);
cInsertEnd(&cSTART1, 350);
cTraverse(cSTART1);

int x;
x=cDeletebeg(&cSTART1);
printf("Deleted element=%d\n", x);
cTraverse(cSTART1);

x=cDeleteEnd(&cSTART1);
printf("Deleted element=%d\n", x);
cTraverse(cSTART1);
int p;
struct node *q;
q=cSTART1;
while(q→info!=200)

```

```

{
q=q→next;
}
cInsertaft(&cSTART1, &q, 460);
cTraverse(cSTART1);
}
/*****

```

- Concatenation of Circular Linked List

ALGORITHM ConcatCircularList(cSTART1, cSTART2)

BEGIN:

p=cSTART1→ Next

cSTART1→ Next=cSTART2→ Next

cSTART2→ Next=p

RETURN cSTRAT2

END;

Time Complexity: $\Theta(1)$

The number of statements executed are 4 which is a constant

Space Complexity: $\Theta(1)$

One extra variable is used namely p.

13.11. DOUBLY LINKED LIST

13.11.1. Primitive Operations

Doubly Linked List, as the name suggests, will keep the address of left and right nodes. Each node will have three fields out of which two are the address fields. dSTART keeps the address of the first node. Initialization is done with dSTART=NULL.

- **Insertion at the Beginning:** A total of 2 pointer adjustment in the node is required.

ALGORITHM InsertBeginning(dSTART, key)

BEGIN:

p=GetNode()

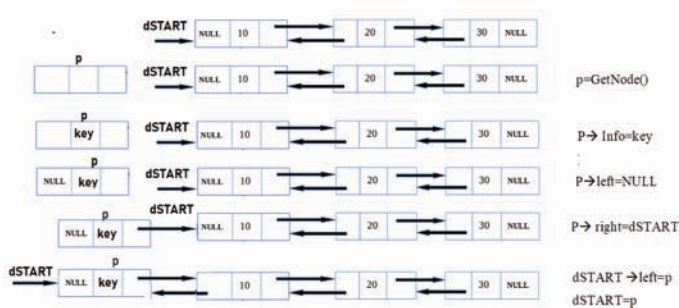
p→ Info=key

p→ Left=NULL

```

p → Right = dSTART
IF dSTART ≠ NULL THEN
dStart → Left = p
dSTART = p
END;

```



Time Complexity: $\Theta(1)$

The number of statements executed are 5 when the list is not Empty and 7 when the list is Empty initially

Space Complexity: $\Theta(1)$

One extra variable is used namely p. Also, the space is allocated in the memory to the new node

- Insertion at the End:** This requires traversal of the linked list till the last node. Newly created node is inserted after the last node. 2 pointer adjustment for linking new node with the last node is required. Along with this the Right field of the new node is set to NULL.

ALGORITHM InsertEnd(dSTART, key)

BEGIN:

```

p = dSTART
q = GetNode()
q → Info = key
IF dSTART == NULL THEN
    InsertBeginning(dSTART, key)
ELSE
    WHILE p → Right ≠ NULL
    p = p → Right
    p → Right = q

```

```
q→Right=NULL
```

```
q→Left=p
```

```
END;
```

Time Complexity: $\Theta(N)$

To Insert in the end of the list, there is a while loop which iterates n times to reach the last element

Space Complexity: $\Theta(1)$

Here two extra variables namely p & q are used. Also, the space is allocated in the memory to the new node

- **Insertion After.Before a Given Node:** A total of 4 pointer adjustment for insertion in the middle of the linked list.

Deletions from the Linked List

ALGORITHM InsertLeft(p , key)

BEGIN:

```
IF  $p \rightarrow \text{Left} = \text{NULL}$  THEN
```

```
    InsertBeginning( $p$ , key)
```

```
ELSE
```

```
IF  $p \rightarrow \text{Left} \neq \text{NULL}$  THEN
```

```
     $q = \text{GetNode}()$ 
```

```
     $r = p \rightarrow \text{Left}$ 
```

```
     $q \rightarrow \text{Info} = \text{key}$ 
```

```
     $q \rightarrow \text{Left} = r$ 
```

```
     $\text{Right}(r) = q$ 
```

```
     $q \rightarrow \text{Right} = p$ 
```

```
     $p \rightarrow \text{Left} = q$ 
```

```
ELSE
```

```
    InsertBeginning( $p$ , key)
```

END;

Time Complexity: $\Theta(1)$

A fixed number of statements are executed in the above algorithm which are linearly independent from the number of elements in the list

Space Complexity: $\theta(1)$

One extra variable **q** is used. Also, the space is allocated in the memory to the new node

ALGORITHM InsertRight(p, key)**BEGIN:****q**=GetNode()**q**→ Info=key**r**=**p**→ Right**p**→ Right=**q****q**→Left=**p**IF **p**→ Right!=NULL THEN**q**→Right=**r**Left(**r**)=**q**

ELSE

q→Right=NULL**END;****Time Complexity:** $\theta(1)$

Here, 7 or 8 statements are executed in any case. Hence, the complexity is considered to be constant.

Space Complexity: $\theta(1)$

Two extra variables **p** and **q** are used here. Also, the space is allocated in the memory to the new node

Deletions from the Liked List

- **Deletion from the Beginning:** dSTART is made to point to second node. In case list has only 1 node, dSTART is set to NULL after this deletion

ALGORITHM Delbeg(dSTART)**BEGIN:**

IF dSTART == NULL THEN

WRITE("void deletion")

```

EXIT(1)
ELSE
p=dSTART
dSTART=dStart→ Right
IF p→ Right!=NULL THEN
dStart→ Left=NULL
x=p→ Info
FreeNode(p)
    RETURN x
END;

```

Time Complexity: $\Theta(1)$

Three statements are executed in case the list is Empty. Else 8 statements are executed. Hence, the complexity is considered to be constant.

Space Complexity: $\Theta(1)$

p is used as an extra variable

- **Deletion at the End:** Linked list is required to be traversed in order to reach the last node and second last node. Next of second last node is made to set NULL. dSTART now keeps the address of second last node.

ALGORITHM DeleteEnd(dSTART)**BEGIN:**

```

IF dSTART == NULL
WRITE("void deletion")
EXIT(1)
ELSE
p=dSTART
q=NULL
WHILE p→ Right!=NULL DO
    q=p
    p=p→ Right
    IF q == NULL THEN
        dSTART=NULL
    ELSE

```

```
q→Right=NULL
x=p→ Info
FreeNode(p)
RETURN x
END;
```

Time Complexity: $\Theta(N)$

The dominating factor, i.e., the while loops iterates n times to reach the end of the list where n is the number of elements in the list where it executes four statements in each iteration.

Space Complexity: $\Theta(1)$

Two extra variables **p** & **q** are used

- **Deletion After/Before a Given Node:** A Total of 2 pointer adjustment are required for this operation.

ALGORITHM DeleteLeft(p)**BEGIN:**

IF $p == \text{NULL} \parallel p \rightarrow \text{Left} == \text{NULL}$ THEN

WRITE(“void deletion”)

EXIT(1)

ELSE

$q = p \rightarrow \text{Left}$

$r = q \rightarrow \text{Left}$

$\text{Right}(r) = p$

$p \rightarrow \text{Right} = r$

$x = q \rightarrow \text{Info}$

$\text{FreeNode}(q)$

RETURN x

END;**ALGORITHM DeleteRight(p)****BEGIN:**

IF $p == \text{NULL} \parallel p \rightarrow \text{Right} == \text{NULL}$ THEN

WRITE(“void deletion”)


```
EXIT(1)
ELSE
q=q→Right
r=q→Right
p→Right=r
Left(r)=p
x=q→Info
FreeNode(q)
RETURN x
END;
```

Time Complexity: $\Theta(1)$

Both the above Algorithms have fixed no of statements independent of the size of the linked list.

Space Complexity: $\Theta(1)$

q & r is used as an extra variable

Traversal**ALGORITHM Traverse(dSTART)****BEGIN:**

```
p=dSTART
WHILE(p!=NULL) DO
WRITE(p→Info)
p=p→Right
END;
```

Time Complexity: $\Theta(N)$

The while loop runs N iterations where the number of nodes is N in the linked list. In each iteration, two statements are executed.

Space Complexity: $\Theta(1)$

An extra variable p is used

C-Program

```
/******  
#include<stdio.h>  
#include<stdlib.h>  
/******  
  
struct node  
{  
int info;  
struct node *Left;  
struct node *Right;  
};  
/******  
  
struct node*GetNode()  
{  
struct node *p;  
p=(struct node*)malloc(sizeof(struct node));  
return p;  
}  
/******  
  
void dInsertBeg(struct node **dSTART, int x)  
{  
struct node *p;  
p=GetNode();  
p→info=x;  
p→Left=NULL;  
p→Right>(*dSTART);  
if(*dSTART!=NULL)  
    (*dSTART)→Left=p;  
(*dSTART)=p;  
}  
/******  
  
void dTraverse(struct node *dSTART)  
{
```

```

if(dSTART == NULL)
{
printf("VOID DELETION");
exit(1);
}
else
{
struct node *p;
p=dSTART;
while(p!=NULL)
{
printf("%d ", p→info);
p=p→Right;
}
}
}
/*****/
void InsertLeft(struct node **p, int key)
{
if((*p)→Left == NULL)
{
dInsertBeg(&(*p), key);
}
else
{
struct node *q, *r;
q=GetNode();
q→info=key;
r=(*p)→Left;
(*p)→Left=q;
q→Right=(*p);
r→Right=q;
q→Left=r;
}
}

```

```
}
}
/*****/
void InsertRight(struct node **p, int key)
{
    struct node *q, *r;
    if((*p)→Right!=NULL)
    {
        q=GetNode();
        q→info=key;
        r=(*p)→Right;
        (*p)→Right=q;
        (q)→Left=(*p);
        (q)→Right=r;
        (r)→Left=q;
    }
    else
    {
        (q)→Left=(*p);
        (*p)→Right=q;
        (q)→Right=NULL;
    }
}
/*****/
int DelBeg(struct node **dSTART)
{
    if((*dSTART) == NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else
```

```

{
    struct node *p;
    p=(*dSTART);
    (*dSTART)=(*dSTART)→Right;
    int x=p→info;
    if(p→Right!=NULL)
    {
        (*dSTART)→Left=NULL;
    }
    free(p);
    return x;
}
}
/*****/
int DeleteEnd(struct node **dSTART)
{
    if((*dSTART) == NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else
    {
        struct node *p, *q;
        p=(*dSTART);
        q=NULL;
        while(p→Right!=NULL)
        {
            p=p→Right;
            q=p→Left;
        }
        if(q == NULL)
        {
            (*dSTART)=NULL;

```

```

    }
    else
    {
        q→Right=NULL;
    }
    int x=p→info;
    free(p);
    return x;
}
}
/*****/
int Deleteatp(struct node **p)
{
    if((*p) == NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else
    {
        int x;
        struct node *r, *q;
        {
            if((*p)→Left == NULL)
            {
                x=DelBeg(&(*p));
            }
            else
            {
                if((*p)→Right == NULL)
                {
                    ((*p)→Left)→Right=NULL;
                }
                else

```

```

    {
        r=(*p)→Left;
        q=(*p)→Right;
        r→Right=q;
        q→Left=r;
    }
    x=(*p)→info;
    free(*p);
}
return x;
}
}
}
/*****/
int DeleteLeft(struct node **p)
{
    if((*p) == NULL || (*p)→Left == NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else
    {
        int x;
        struct node *r, *q;
        r=(*p)→Left;
        if(r→Left == NULL)
        {
            (*p)→Left=NULL;
        }
        else
        {
            q=r→Left;
            q→Right=(*p);

```

```
    (*p)→Left=q;
}
x=r→info;
free(r);
return x;
}
}
/*****/
int DeleteRight(struct node **p)
{
    if((*p) == NULL || (*p)→Right == NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else
    { int x;
      struct node *r, *q;
      r=(*p)→Right;
      if(r→Right == NULL)
      {
          (*p)→Right=NULL;
      }
      else
      {
          q=r→Right;
          q→Left=(*p);
          (*p)→Right=q;
      }
      x=r→info;
      free(r);
      return x;
    }
}
```



```

/*****/
void dInsertEnd(struct node **dSTART, int key)
{
    struct node *p;
    p=(*dSTART);
    if(p == NULL)
    {
        dInsertBeg(&(*dSTART), key);
    }
    else
    {
        struct node *q;
        while(p→Right!=NULL)
        {
            p=p→Right;
        }
        q=GetNode();
        p→Right=q;
        q→info=key;
        q→Left=p;
        q→Right=NULL;
    }
}

/*****/

int main()
{
    struct node *dSTART1;
    dSTART1=NULL;
    dInsertBeg(&dSTART1, 300);
    dInsertBeg(&dSTART1, 200);
    struct node *p;
    p=dSTART1;
    dInsertBeg(&dSTART1, 100);
    dInsertBeg(&dSTART1, 500);
}

```

```
dInsertBeg(&dSTART1, 1600);
dInsertBeg(&dSTART1, 700);
dInsertBeg(&dSTART1, 800);
dInsertBeg(&dSTART1, 900);
dInsertEnd(&dSTART1, 400);
dTraverse(dSTART1);
```

```
printf("\n");
dInsertBeg(&dSTART1, 90);
dTraverse(dSTART1);
```

```
printf("\n");
InsertLeft(&p, 150);
dTraverse(dSTART1);
```

```
int x;
InsertRight(&p, 150);printf("\n");
dTraverse(dSTART1);
```

```
x=DeleteLeft(&p);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
```

```
x=DeleteRight(&p);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
```

```
x=Deleteatp(&p);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
```

```
x=DelBeg(&dSTART1);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
```

```

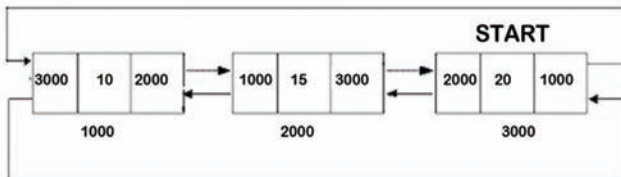
x=DeleteEnd(&dSTART1);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
}
/*****

```

13.12. CIRCULAR DOUBLY LINKED LIST

This linked list is much similar to Circular linked list. The identification is through the address of the last node. The number of pointer adjustments are double of that of the circular linked list.

Here the left field of the first node keeps the address of the last node.



13.12.1. Primitive Operations

- Insertion at the Beginning

ALGORITHM InsertEnd(dSTART, key)

BEGIN:

p=dSTART→Next

q=GetNode()

q→Info=key

IF(dSTART == NULL) THEN

q→left=q

q→right q

ELSE

q→Right=p

p→Left=q

q→Left=dSTART

dSTART→Left=q

END;

Time Complexity: $\Theta(1)$

Three statements at the Beginning unconditionally. 2/4 statements are conditional. Total statements are constant irrespective of the size of the linked list

Space Complexity: $\Theta(1)$

Two extra variables are used, namely **p** & **q**. Also, the space is allocated in the memory to the new node

- Insertion at the end

ALGORITHM InsertEnd(dSTART, key)

BEGIN:

IF dSTART == NULL THEN

 InsertBeginning(dSTART, key)

ELSE

p=dSTART→Next

q=GetNode()

q→Info=key

 q→Right=p

p→Left=q

p→Left=dSTART

dSTART→Right=q

dSTART=q

END;

Time Complexity: $\Theta(1)$

The number of statements executed are 8 when the list is not Empty and A call to InsrBeginning if list is Empty ensures constant number of instructions

Space Complexity: $\Theta(1)$

Two extra variables are used namely **p** & **q**. Also, the space is allocated in the memory to the new node

ALGORITHM InsertLeft(p, key)**BEGIN:**

IF p == NULL THEN

WRITE(“\void Insertion”)

ELSE

IF p → Left == NULL THEN

InsertBeginning()

ELSE

q=GetNode()

q → Left = p → Left

q → Right = p

q → Info = key

Right(p → Left) = q

END;**Time Complexity: $\theta(1)$**

A fixed number of statements are executed in the above algorithm which are linearly independent from the number of elements in the list

Space Complexity: $\theta(1)$

p & q are used as variables

ALGORITHM InsertRight(p, key)**BEGIN:**

IF p == NULL THEN

InsertBeginning()

ELSE

q=GetNode()

q → Left = p

q → Right = p → Right

p → Right → Left = q

p → Right = q

END;

Time Complexity: $\Theta(1)$

In case p is null, InsertBeginning is called whose complexity is $\Theta(1)$. Else, six statements are executed

Space Complexity: $\Theta(1)$

p & q are used as variables

ALGORITHM DeleteBeginning(dSTART)**BEGIN:**

IF dSTART=NULL THEN

 WRITE("void deletion")

 EXIT(1)

ELSE

p=dSTART → Right

IF dSTART → Right == dSTART THEN

dSTART=NULL

ELSE

q=p → Right

q → Left=dSTART

dSTART → Right=q

 x=p → Info

FreeNode(p)

RETURN x

END;**Time Complexity: $\Theta(1)$**

A fixed number of statements are executed in the above algorithm which is linearly independent from the number of elements in the list

Space Complexity: $\Theta(1)$

p, x & q are used as variables

ALGORITHM DeleteEns(dSTART)**BEGIN:**

IF dSTART=NULL THEN

WRITE(“\void deletion”)

ELSE

p=dSTART

IF dSTART→ Right == dSTART THEN

dSTART=NULL

ELSE

q→ Right=dSTART→ Right

(dSTART→ Right)→ Left=q

dSTART=q

x=p→ Info

FreeNode(p)

RETURN x

END;**Time Complexity: $\Theta(1)$**

A fixed number of statements are executed in the above algorithm which are linearly independent from the number of elements in the list

Space Complexity: $\Theta(1)$

p, x & q are used as variables

PROGRAM:-

```

/*****
#include<stdio.h>
#include<stdlib.h>
/*****

struct node
{
int info;
```

```

struct node *Left;
struct node *Right;
};
/*****/

struct node*GetNode()
{
struct node *p;
p=(struct node*)malloc(sizeof(struct node));
return p;
}
/*****/

void dInsertBeg(struct node **cdSTART, int x)
{
struct node *p;
p=GetNode();
p→info=x;
if((*cdSTART) == NULL)
{
p→Right=p;
p→Left=p;
(*cdSTART)=p;
}
else
{
struct node *q;
q=(*cdSTART)→Right;
p→Right=q;
q→Left=p;
p→Left=(*cdSTART);
(*cdSTART)→Right=p;
}
}
/*****/

void dTraverse(struct node *dSTART)

```



```

{
if(dSTART == NULL)
{
    printf("Empty list\n");
    exit(1);
}
else
{
    struct node *p, *q;
    p=(dSTART)→Right;
    q=(dSTART)→Left;
    while(q!=(dSTART))
    {
        printf("%d ", p→info);
        p=p→Right;
        q=p→Left;
    }
    printf("\n");
}
}

/*****/

void InsertLeft(struct node **p, int key)
{
    if((*p) == NULL)
    {
        printf("Insertion not possible");
        exit(1);
    }
    else
    {
        struct node *q, *r;
        r=GetNode();
        r→info=key;
        if((*p)→Right == (*p))

```

```

{
    (*p)→Left=r;
    r→Right=(*p);
    r→Left=(*p);
    (*p)→Right=r;
}
else
{q=(*p)→Left;
 r→Right=(*p);
 r→Left=q;
 (*p)→Left=r;
 q→Right=r;

}
}
}
/*****/
void InsertRight(struct node **p, int key)
{
    if((*p) == NULL)

    {
        printf("Insertion not possible");
        exit(1);
    }
    else
    {
        struct node *q, *r;
        r=GetNode();
        r→info=key;
        if((*p)→Right == (*p))
        {
            (*p)→Right=r;
            r→Left=(*p);
            r→Right=(*p);

```

```

    (*p)→Left=r;
}
else
{q=(*p)→Right;
r→Left=(*p);
r→Right=q;
(*p)→Right=r;
q→Left=r;

}
}}
/*****/

int DelBeg(struct node **dSTART)
{
    if((*dSTART) == NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else
    {
        int x;
        if((*dSTART)→Right == (*dSTART))
        {
            struct node *p;
            p=(*dSTART);
            (*dSTART)=NULL;
            x=p→info;
            return x;
        }
        else
        {
            struct node *p, *q;
            q=(*dSTART)→Right;

```

```
x=q→info;
p=q→Right;
(*dSTART)→Right=p;
p→Left=(*dSTART);
free(q);
return x;
}

}}
/*****/
int DeleteEnd(struct node **dSTART)
{
    if((*dSTART) == NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else
    {
        int x;
        if((*dSTART)→Right == (*dSTART))
        {
            struct node *p;
            p=(*dSTART);
            (*dSTART)=NULL;
            x=p→info;
            return x;
        }
        else
        {
            struct node *p, *q;
            q=(*dSTART)→Right;
            p=(*dSTART)→Left;
            p→Right=q;
```

```

    q→Left=p;
    x=(*dSTART)→info;
    (*dSTART)=p;
    return x;
}

}}
/*****/
int Deleteatp(struct node **p)
{
    if((*p) == NULL)
    {
        printf("void deletion\n");
        exit(1);
    }
    else
    { int x;
      struct node *r, *q;
      if((*p)→Left == (*p))
      {
          x=DelBeg(&(*p));
      }
      else
      {
          r=(*p)→Left;
          q=(*p)→Right;
          r→Right=q;
          q→Left=r;
          x=(*p)→info;
          free(*p);
      }
      return x;
    }
}

```

```
/******
```

```
int DeleteLeft(struct node **p)
```

```
{
    if((*p) == NULL || (*p)→Left == (*p))
    {
        printf("void deletion\n");
        exit(1);
    }
    else
    { int x;
      struct node *r, *q;
      r=(*p)→Left;
      if(r→Left == (*p))
      {
          (*p)→Left=(*p);
      }
      else
      {
          q=r→Left;
          q→Right=(*p);
          (*p)→Left=q;
      }
      x=r→info;
      free(r);
      return x;
    }
}
```

```
/******
```

```
void dInsertEnd(struct node **dSTART, int key)
```

```
{
    struct node *p;
    p=(*dSTART);
    p=GetNode();
    p→info=key;
```

```

if(p == NULL)
{
dInsertBeg(&(*dSTART), key);
}
else
{
    struct node *q;
    q=(*dSTART)→Right;
    (*dSTART)→Right=p;
    p→Left=(*dSTART);
    p→Right=q;
    q→Left=p;
    (*dSTART)=p;
}
}
/*****/

int main()
{
struct node *dSTART1;
dSTART1=NULL;
dInsertBeg(&dSTART1, 300);
dInsertBeg(&dSTART1, 200);
dInsertBeg(&dSTART1, 100);
struct node *p;
p=dSTART1;
dTraverse(dSTART1);
dInsertEnd(&dSTART1, 400);
dTraverse(dSTART1);
InsertRight(&p, 150);
printf("\n");
dTraverse(dSTART1);
printf("\n");
InsertLeft(&p, 150);
dTraverse(dSTART1);

```

```
int x;
x=DelBeg(&dSTART1);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
x=DeleteEnd(&dSTART1);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
x=DeleteLeft(&p);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
x=Deleteatp(&p);
printf("\nDeleted element %d\n", x);
dTraverse(dSTART1);
}
/*****/
```


EXERCISES

- 1) Explain the concept of header linked list
- 2) Explain the concept of Generalized linked list
- 3) Complexity of insert-end operation with n nodes in the linear linked list is

a. $O(1)$

b. $O(n^2)$

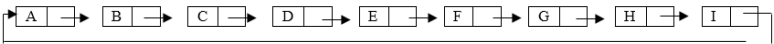
c. $O(\log_2 n)$

d. $O(n)$

- 4) The polynomial represented by the linked list poly is

		Coefficient	exponent	Next
Poly Avail	0	3	4	7
	1	5	7	5
	2			4
	3	-5	1	6
	4			
	5	-9	5	0
	6	-4	0	-1
	7	8	2	3

- a. $5x^7 - 9x^5 + 3x^4 + 8x^2 - 5x - 4$
- b. $5x^7 + 9x^5 + 3x^4 + 8x^2 + 5x + 4$
- c. $5x^7 - 9x^5 - 3x^4 - 8x^2 - 5x - 4$
- d. $5x^7 - 9x^5 - 3x^4 + 8x^2 - 5x - 4$
- 5) The only node that remains in the last for the Josephus problem with count 7 is.



- a. B
- b. D
- c. A
- d. I
- 6) The complexity of deletion of the last node from a doubly linked list is

a. $O(1)$

b. $O(n)$

- c. $O(n^2)$
- d. $O(n \log_2 n)$

7) No of nodes in the answer linked list if the two numbers given below are added (group the numbers in 4 digit each)

99999912345678912345678997812

123456789123456789123451234

- a. 7
- b. 8
- c. 6
- d. 9

8) Consider the following statement about the linked list

- a. Global data structure
- b. linear data Structure
- c. traversal requires $O(n)$ time
- i. a & b are true whereas c is false
- ii. all a, b & c are true
- iii. b & c are true and a is false
- iv. only b is true

9) What is the purpose of using header nodes in the linked?

- a. To contain global information about the list
- b. To contain the pointer to the last node
- c. To collect the deleted nodes
- d. None of these

10) Which among the following function will require the concept of previous node pointer (for linear linked list)

- a. Ordered insertion
- b. Deletion of all occurrences of x
- c. Insertion in the end
- d. all of the above

11) Comparisons required to insert a node after the last node in a circular linked list is

- a. $O(\log n)$
- b. $O(n)$
- c. $O(1)$
- d. None of the above

12) Minimum no of pointer adjustment for deletion of last node from doubly linked list is

- a. 1
- b. 2
- c. 3

- d. 4
13. If a circular linked list of size n were broken from the mid to form two different linked list of approximately the same length, effort required for the same will be
- $O(n)$
 - $O(1)$
 - $O(n \log n)$
 - $O(n^2)$
14. In the linked representation of polynomials, one header node is also used that contains invalid information. If the structure of a node is as given below, what will be the value of Coef & Exp part?

Coef	Exp	Next
------	-----	------

15. Given an ordered circular list of size n . Insertion of x at the appropriate position in the list requires Time
- $O(n)$
 - $O(\log n)$
 - $O(1)$
 - None of these
16. If two polynomials of degree one & n terms are represented in the form of linked list, the multiply operation of these polynomials requires Time
- $O(n)$
 - $O(n^3)$
 - $O(n \log n)$
 - $O(n^2)$
17. Which among the following difference is not correct between array and Linked list
- Insertion in array is costlier whereas in Linked list is cheaper
 - Deletion in array is costlier whereas in Linked list is cheaper
 - Traversal in array is costlier whereas in Linked list is cheaper
 - Linked list requires more memory than array
18. An ascending priority queue is implemented using linked list. Which of the following operation is cheapest
- Traversal
 - Insertion
 - Delete-min
 - Search

-
- 19. A Descending priority queue is implemented using linked list. Which of the following operation is cheapest**
- e. Traversal
 - f. Insertion
 - g. Search
 - h. Delete-Max

Programming Questions

- 1. Write the C-code for implementation of Linked List using array.
- 2. Write a C-code for splitting the given linked list from mid
- 3. Write a C-code for deletion of all occurrences of x from linked list

14

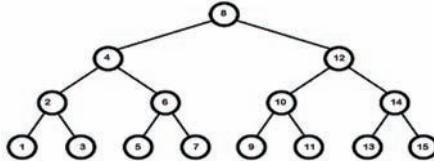
Binary Tree

CONTENTS

14.1. Binary Tree Basics.....	338
14.2. Implementation Of Binary Tree.....	340
14.3. Primitive Operations In Binary Tree.....	341
14.4. Building Binary Tree From Given Traversals.....	347
14.5. Level Order Traversal In Binary Tree.....	348
14.6. Application Problems Related To Binary Tree	349
14.7. Expression Tree	356
14.8. Huffman Coding.....	358
Exercises.....	360

14.1. BINARY TREE BASICS

These are a special kind of trees where each node can have a maximum of two children. As a definition, The tree elements are treated as a set. This set can be partitioned into three disjoint subsets. One of which is the root. Other two are themselves the binary trees called left and right subtrees.



In the above tree, $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$

$S_1 = \{8\}$

$S_2 = \{1, 2, 3, 4, 5, 6, 7\}$

$S_3 = \{9, 10, 11, 12, 13, 14, 15\}$

S_1 is root

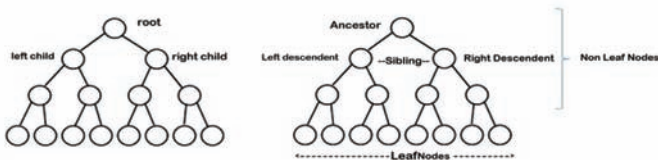
S_2 is Left Subtree

S_3 is Right Subtree

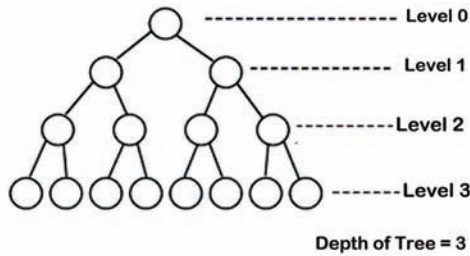
Binary Tree are used in various computer science applications e.g.,

- Data compression through Huffman coding;
- Representation of arithmetic expression;
- Searching (using binary search tree (BST));
- Sorting (using heap);
- Making decision trees etc.

14.1.1. Relation of Nodes in a Binary Tree



- **Strictly Binary Tree/2-Tree/Full Tree:** All the nodes in the tree will either have 2 child or 0 child. Nodes with only one child is not allowed.
- **Complete Binary Tree:** This is a strictly binary tree with all the leaf nodes at the same level.



Total no of nodes in such a tree are:

$N = \text{Nodes at level 0} + \text{Nodes at level 1} + \text{Nodes at level 2} + \dots + \text{Nodes at level } d$
 $= 1 + 2^1 + 2^2 + 2^3 + \dots + 2^d$ (A geometric progression with common ratio d and $d+1$ terms)

$$= 1 * (2^{d+1} - 1) / (2 - 1)$$

$$N = (2^{d+1} - 1)$$

If we readjust the above equality,

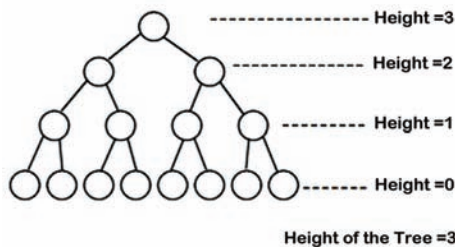
$$N + 1 = 2^{d+1}$$

$$d + 1 = \log_2(N + 1)$$

$$d = \log_2(N + 1) - 1$$

Above equality shows that we can find out the depth of the complete binary tree if no of nodes are given

- **Height of the Nodes in the Binary Tree:** Height of a node is the distance of a node from the leaf node. e.g.,



Height of the root node is called the height of the tree. Depth of the tree is same as height of the tree.

The height of a binary tree is least if the tree is Complete. Height will increase in case the tree is not full. If all the nodes of the tree are on one side only, the height may go up to $N-1$. This type of tree is called Skewed Binary Tree.

- **Finding number of Binary Trees with Given Nodes:** If there is one node, only 1 tree is possible

If there are two nodes, 2 trees are possible.

If there are 3 nodes, total trees can be 5



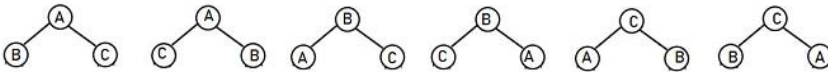
No of possible trees can be given by the formula:

$$2^N C_N / ((N+1))$$

For 3 nodes

$${}^6C_3 / 4 = 5$$

If information is given, e.g., A, B, C in nodes, total trees possible are:



6 tree of the same structure is possible. i.e., 3! Hence total tree possible can be given by the formula:

$$({}^{2^N}C_N / (N+1)) * N!$$

For 3 nodes A, B, C this figure is 30

14.2. IMPLEMENTATION OF BINARY TREE

Binary tree can be implemented using:

- Array;
- Linked list.

14.2.1. Array Implementation

This type of implementation suits only complete or almost complete binary trees. This implementation is also called implicit binary tree representation.

In this representation if we consider Array indexes starting from 1,

If parent is at I index,

Left Child: $2*I$ Index

Right Child: $2*I+1$ Index

If Child at I index

Parent: $\text{Floor}(I/2)$

If Array index are considered to start from 0,

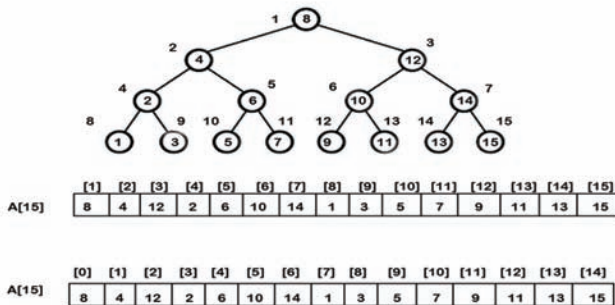
If parent is at I index,

Left Child: $2*I+1$ Index

Right Child: $2*I+2$ Index

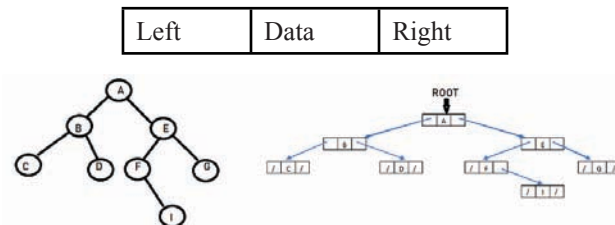
If Child at I index

Parent: $\text{Floor}((I-1)/2)$



14.2.2. Linked List Implementation

In this linked implementation, each node consists of at least 3 fields. This representation is done through doubly linked list.



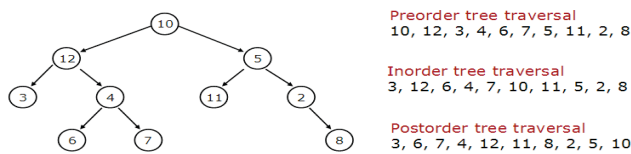
14.3. PRIMITIVE OPERATIONS IN BINARY TREE

- Traversal:** Out of the tree information associated with a node, i.e., Data (D), Left (L) and Right (R), there are six combination possible for this 3

Out of these six, the combination in which left (L) written first and right (R) next, are selected.

Combination	Selection/Rejection	Method	
DLR	Selected	Data is first	Pre Order Traversal
DRL	X		
LDR	Selected	Data is in mid	In Order Traversal
LRD	Selected	Data is at the last	Post Order Traversal
RDL	X		
RLD	X		

All three Traversal Methods: Pre-Order (DLR), In-Order (LDR) and Post-Order (LRD) are recursive in nature.



14.3.1. Pre Order Traversal

ALGORITHM PreorderTraversal(root)

BEGIN:

IF root != NULL THEN

WRITE(Data(root))

PreorderTraversal(Left(root))

PreorderTraversal(Right(root))

END;

Time Complexity: $\Theta(N)$

As every Node is printed once

Space Complexity: $(\log_2 N), O(N)$

If the tree is balanced, Call stack will have $\log_2 n+1$ entries to the maximum at any moment in case the tree is balanced. If the tree is skewed then the call stack can be grow up to n activation records

14.3.2. In Order Traversal

ALGORITHM InorderTraversal(root)

BEGIN:

WHILE root != NULL THEN

InorderTraversal(Left(root))

```
WRITE(Data(root))  
InorderTraversal(Right(root))  
END;
```

Time Complexity: $\Theta(N)$

As every Node is printed once

Space Complexity: $(\log_2 N)$, $O(N)$

If the tree is balanced, Call stack will have $\log_2 n + 1$ entries to the maximum at any moment in case the tree is balanced. If the tree is skewed then the call stack can be grow up to n activation records

14.3.3. Post Order Traversal

ALGORITHM PostorderTraversal(root)

BEGIN:

```
IF root != NULL THEN  
PostorderTraversal(Left(root))  
PostorderTraversal(Right(root))  
WRITE(Data(root))  
END;
```

Time Complexity: $\Theta(N)$

As every Node is printed once

Space Complexity: $(\log_2 N)$, $O(N)$

If the tree is balanced, Call stack will have $\log_2 n + 1$ entries to the maximum at any moment in case the tree is balanced. If the tree is skewed then the call stack can be grow up to n activation records

14.3.4. Creation of Binary Tree

It is a recursive procedure which asks the user to enter the data value of nodes and its left and right nodes.

ALGORITHM CreateTree(tree)

BEGIN:

```
WRITE("Whether Left of DATA(tree) exists (1/0)")  
READ(choice)
```

```

IF (choice == 1) THEN
WRITE("Input the information of Left node")
READ(x)
p ← MakeNode(x)
LEFT(tree) ← p
CreateTree(p)

WRITE("Whether Right of DATA(tree) exists (1/0)")
READ(choice)

IF (choice == 1) THEN
WRITE("Input the information of Right node")
READ(x)
p ← MakeNode(x)
Right(tree) ← p
CreateTree(p)
END;

```

Time Complexity: $\Theta(N)$

There are N elements for creating nodes.

Space Complexity: $\Theta(N)$

There are N nodes created

C-Program

```

/*****/
#include<stdio.h>
#include<stdlib.h>
/*****/

struct node
{
    int data;
    struct node *Left;

```

```
struct node *Right;
};
/*****/
struct node *MakeNode(int x)
{
    struct node *p;
    p=(struct node*)malloc(sizeof(struct node));
    p→data=x;
    p→Left=NULL;
    p→Right=NULL;
    return p;
}
/*****/
void PreOrderTraversal(struct node *root)
{
    if(root!=NULL)
    {
        printf(“ %d “, root→data);
        PreOrderTraversal(root→Left);
        PreOrderTraversal(root→Right);
    }
}
/*****/
void PostOrderTraversal(struct node *root)
{
    if(root!=NULL)
    {
        PostOrderTraversal(root→Left);
        PostOrderTraversal(root→Right);
        printf(“ %d “, root→data);
    }
}
/*****/
void InOrderTraversal(struct node *root)
```

```
{
if(root!=NULL)
{
InOrderTraversal(root→Left);
printf(“ %d “, root→data);
InOrderTraversal(root→Right);
}
}

void CreateTree(struct node **t)
{
int choice, x;
struct node *p;
printf(“whether Left of %d exits(1/0)\t”, (*t)→data);
scanf(“%d”, &choice);
if(choice == 1)
{
printf(“input the key of Left node of root\t”);
scanf(“%d”, &x);
p=MakeNode(x);
(*t)→Left=p;
CreateTree(&p);
}
printf(“whether Right of %d exits(1/0)\t”, (*t)→data);
scanf(“%d”, &choice);
if(choice == 1)
{
printf(“input the key of Right node of root\t”);
scanf(“%d”, &x);
p=MakeNode(x);
(*t)→Right=p;
CreateTree(&p);
}
}
```

```

/*****/
int main()
{
    struct node *root;
    int x;
    printf("input the key of root\t");
    scanf("%d", &x);
    root=MakeNode(x);
    CreateTree(&root);
    printf("\npre order traversal \n");
    PreOrderTraversal(root);
    printf("\npost order traversal\n");
    PostOrderTraversal(root);
    printf("\nin order traversal \n");
    InOrderTraversal(root);
    printf("\n");
}
/*****/

```

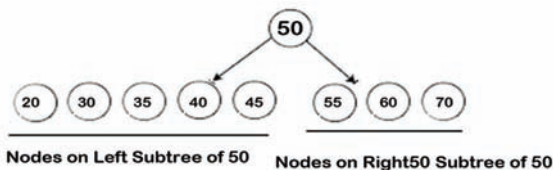
14.4. BUILDING BINARY TREE FROM GIVEN TRAVERSALS

Binary tree can be built uniquely if combination of pre order & in order or post order & in order traversals are given.

If combination of pre order & in order is given

inorder = {20, 30, 35, 40, 45, 50, 55, 60, 70}

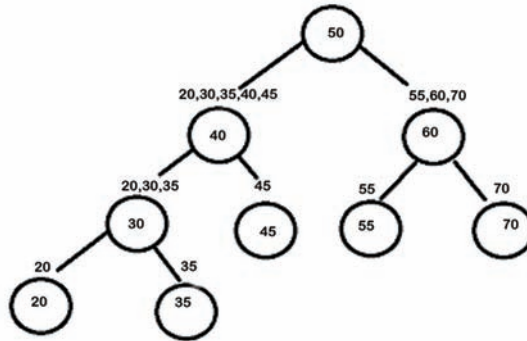
preorder = {50, 40, 30, 20, 35, 45, 60, 55, 70}



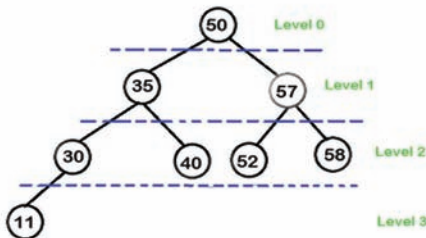
First Node in Pre Order Traversal is the root node. Find that node in In Order.

All nodes on the left of this node in Inorder traversal are the left subtree nodes and All nodes on the right of this node in Inorder traversal are the Right subtree nodes. Repeat the same with the next element of Pre order that will further divide the set into two.

Resulting Tree for the example in the above figure is:



14.5. LEVEL ORDER TRAVERSAL IN BINARY TREE



Level Order traversal:

50	35	57	30	40	52	58	11
L0	L1		L2				L3

For performing level order traversal, we require a Queue. First store the root in the queue and till the time queue does not become empty, keep deleting an element from the Queue, print the data of the deleted node and insert its left and right child in the same order in the Queue.

ALGORITHM LevelOrderTraversal(root)

BEGIN:

Queue Q

Initialize(Q)

Enqueue(root)

WHILE !Empty(Q) DO

x ← DeQueue(Q)

```
    WRITE(Data(x))
    IF LEFT(x) !=NULL THEN
        EnQueue(x)
    IF Right(x) !=NULL THEN
        EnQueue(x)
END;
```

Time Complexity: $\Theta(N)$

Each node is inserted in the queue once and deleted from queue once. Total of $2*N$ operations of constant time.

Space Complexity: $\Theta(N)$

Queue size is N

14.6. APPLICATION PROBLEMS RELATED TO BINARY TREE

- Finding Height of Binary Tree

ALGORITHM HEIGHT(root)

BEGIN:

IF root == NULL THEN

RETURN 0

ELSE

IF Left(root) == NULL AND Right(root) == NULL

RETURN 0

ELSE

RETURN 1+Max(HEIGHT(Left(root)), Height(Right(root)))

END;

Time Complexity: $\Theta(N)$

Need to reach all the node once for computing height

Space Complexity: $(\log_2 N), O(N)$

If the tree is balanced, Call stack will have $\log_2 n + 1$ entries to the maximum at any moment in case the tree is balanced. If the tree is skewed then the call stack can be grow up to n activation records

C Function:-

```

/*****/
int height(struct node*root)
{
if(root == NULL)
return 0;
else if(root→Left == NULL && root→Right == NULL)
return 0;
else
return(1+max(height(root→Left), height(root→Right)));
}
/*****/

```

Finding count of nodes having 0 children in Binary Tree

ALGORITHM CountOfN0(root)

BEGIN:

IF tree == NULL THEN

RETURN 0

ELSE

IF LEFT(tree) == NULL && Right(tree) == NULL THEN

RETURN 1

ELSE

RETURN CountOfN0(LEFT(tree))+CountOfN0(Right(tree));

Time Complexity: $\Theta(N)$

Need to reach all the node once for computing height

Space Complexity: ($\log_2 N$), $O(N)$

If the tree is balanced, Call stack will have $\log_2 n + 1$ entries to the maximum at any

moment in case the tree is balanced. If the tree is skewed then the call stack can be grow up to n activation records

C Function

```

/*****
int N0Count(struct node *tree)
{
    if(tree == NULL)
        return 0;
    else
    {
        if((tree->Left == NULL)&&(tree->Right == NULL))
            return 1;
        else
            return (N0Count(tree->Left)+N0Count(tree->Right));
    }
}
*****/

```

Finding count of nodes having 1 children in Binary Tree

ALGORITHM CountOfN1(root)

BEGIN:

IF tree == NULL THEN

RETURN 0

ELSE

IF LEFT(tree) == NULL && Right(tree) == NULL THEN

RETURN 0

ELSE

IF LEFT(tree) != NULL && Right(tree) != NULL THEN

RETURN CountOfN1(LEFT(tree))+CountOfN1(Right(tree));

ELSE

```
RETURN 1+CountOfN1(LEFT(tree))+CountOfN1(Right(tree))
```

Time Complexity: $\Theta(N)$

Need to reach all the node once for computing height

Space Complexity: $(\log_2 N)$, $O(N)$

If the tree is balanced, Call stack will have $\log_2 n + 1$ entries to the maximum at any moment in case the tree is balanced. If the tree is skewed then the call stack can be grow up to n activation records

C Function:

```

/*****/
int N1Count(struct node *tree)
{
    if(tree == NULL)
        return 0;
    else
    {
        if((tree→Left == NULL)&&(tree→Right == NULL))
            return 0;
        else if (tree→Left!=NULL && tree→Right!=NULL)
            return (N1Count(tree→Left)+N1Count(tree→Right));
        else
            return 1+(N1Count(tree→Left)+N1Count(tree→Right));
    }
}
/*****/

```

Finding count of nodes having 2 children in Binary Tree

ALGORITHM CountOfN2(tree)

BEGIN:

IF tree == NULL THEN

 RETURN 0

ELSE

```

        IF LEFT(tree) == NULL && Right(tree) == NULL THEN
RETURN 0
        ELSE
        IF LEFT(tree) != NULL && Right(tree) != NULL THEN
            RETURN 1+ CountOfN1(LEFT(tree))+CountOfN1(Rig
ht(tree))
        ELSE
            RETURN CountOfN1(LEFT(tree))+CountOfN1(Right(t
ree))

```

Time Complexity: $\Theta(N)$

Need to reach all the node once for computing height

Space Complexity: $(\log_2 N)$, $O(N)$

If the tree is balanced, Call stack will have $\log_2 n + 1$ entries to the maximum at any moment in case the tree is balanced. If the tree is skewed then the call stack can be grow up to n activation records

C Function:

```

/*****
int N2Count(struct node *tree)
{
    if(tree == NULL)
        return 0;
    else
    {
        if((tree→Left == NULL)&&(tree→Right == NULL))
            return 0;
        else if (tree→Left!=NULL && tree→Right!=NULL)
            return 1+(N2Count(tree→Left)+N2Count(tree→Right));
    }
    return (N2Count(tree→Left)+N2Count(tree→Right));
}
*****/

```

Finding if the Binary Tree is Complete**ALGORITHM isComplete (root, index, number_nodes)****BEGIN:**

IF tree == NULL THEN

RETURN True;

// If index assigned to current node is more than

// number of nodes in tree, then tree is not complete

IF index >= number_nodes THEN

RETURN False;

// Recursive for Left and Right subtrees

RETURN (isComplete(LEFT(root), 2*index + 1, number_nodes) AND
isComplete(Right(root), 2*index + 2, number_nodes))**END;****Time Complexity: $\Theta(N)$**

Need to reach all the node once for computing height

Space Complexity: $(\log_2 N)$, $O(N)$

If the tree is balanced, Call stack will have $\log_2 n + 1$ entries to the maximum at any moment in case the tree is balanced. If the tree is skewed then the call stack can be grow upto n activation records

C Function

/*****

int NodeCount(struct node* root)

{

if (root == NULL)

return (0);

return (1 + NodeCount(root→Left) + NodeCount(root→Right));

}

*****/

int isComplete (struct node* root, int index, int number_nodes)

{

```

if (root == NULL)
    return 1;

if (index >= number_nodes)
    return 0;

return (isComplete(root→Left, 2*index + 1, number_nodes) &&
        isComplete(root→Right, 2*index + 2, number_nodes));
}
/*****/

```

Finding Balancing factor of a node

LH represents the Left height and RH represents right height. $LH - RH$ is the balance Factor of the given node.

ALGORITHM BalanceFactor(N)

BEGIN:

IF Left(N) != NULL

LH $\leftarrow$ 1 + h(Left(N))

ELSE

LH $\leftarrow$ 0

IF Right(N) != NULL

RH $\leftarrow$ 1 + h(Right(N))

ELSE

RH $\leftarrow$ 0

RETURN(LH – RH)

END;

Time Complexity: $\Theta(N)$

Need to reach all the node once for computing height

Space Complexity: $(\log_2 N)$, $O(N)$

If the tree is balanced, Call stack will have $\log_2 n + 1$ entries to the maximum at any

moment in case the tree is balanced. If the tree is skewed then the call stack can be grow upto n activation records

14.7. EXPRESSION TREE

Expression Tree helps us to evaluate the given Arithmetic/Logic expression. To understand building of expression Tree, we will first learn how to build expression tree for Binary operators. The concept will further be extended to understand building Expression Tree for Unary operators as well.

While building the expression tree, we will observe that:

- All the operators are the non-leaf nodes;
- All the operands are the leaf nodes.

If expression contains only the binary operators, the resulting tree will be 2-Tree/Strictly Binary Tree

How to build an expression tree:

First convert the given expression into postfix

A stack is used for constructing an expression tree. The expression is traversed and following actions taken:

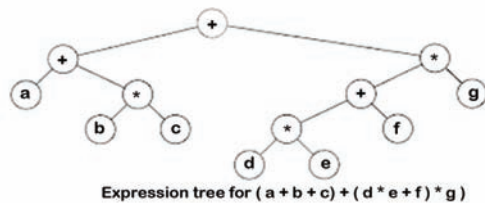
- If the symbol is an operand, push it in the stack;
- If symbol is the operator, pop two items from stack and set them as the right and left child of operator respectively and push the current node again on stack.

Towards the end, Pop the stack. The symbol popped will be root of the given expression.

Consider the Expression $a+b$

Postfix= $ab+$

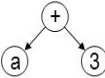
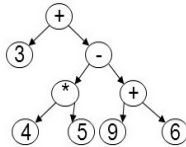
Symbol	Stack
a	a
b	a,b
+	



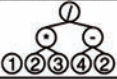
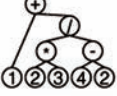
14.7.1. Properties of Expression Tree

The In Order Traversal of Expression Tree should result the infix equivalent of the expression.

If building the expression tree for unary operator, the above property will be very helpful

Expression	Expression Tree	Inorder Traversal
(a+3)		a + 3
3+(4*5-(9+6))		3+4*5-9+6
log(x)		log x
n!		n !

14.7.2. Building Expression Tree and Conversion to Postfix Simultaneously

Expression String	Expression Stack	Operator Stack
1+2*3/(4-2)		
+2*3/(4-2)	①	
2*3/(4-2)	①	+
*3/(4-2)	①②	+
3/(4-2)	①②	÷*
/(4-2)	①②③	÷*
(4-2)		÷/
4-2)		÷/(
-2)	①②③④	÷/(
2)		÷/(-
)	①②③④②	÷/(-
		÷/
		÷
		

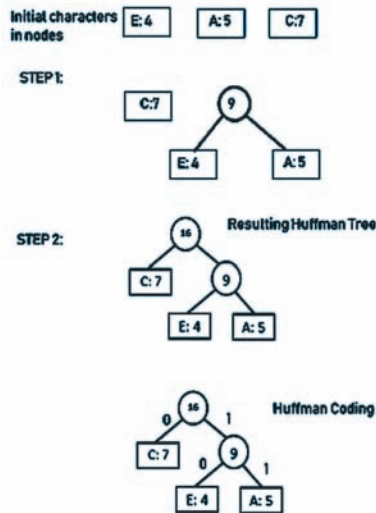
14.8. HUFFMAN CODING

Huffman Coding is the Lossless text compression Technique. It generates the variable length codes for various characters appearing in the text. Each of these codes are unique prefix codes that makes decoding straight forward.

The coding mechanism counts the frequency of various characters appearing in the text or their occurrence probability. The Algorithm builds Huffman tree using the Priority Queue.

14.8.1. Building Huffman Tree

The example of building Huffman Tree and Codes are given below

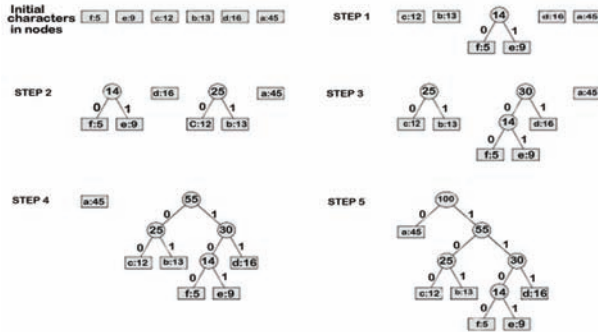


Codes for various characters in the figure.

C: 0 E: 10 A: 11

For the character that appears more times, its code is shorter and the characters which appear less number of times, its code is longer.

Example 2:



A Priority Queue is created with insertion of all N characters (these characters are converted to nodes before insertion) For N characters, the $N-1$ steps are performed. In each step following things happen:

- Two items are deleted from Priority queue as X and Y .
- New node Z is created with data value $X \rightarrow \text{Data} + Y \rightarrow \text{Data}$.
- X is set as Left of Z .
- Y is set as Right of Z .
- Z is inserted in the Priority Queue.

For Coding, left branch is given 0 code and right 1 for each branch created.

ALGORITHM HuffmanTree($C[]$, N)

BEGIN:

PriorityQueue PQ

PQ $\leftarrow$ NULL

FOR $i \leftarrow 1$ to N DO

$X \leftarrow \text{MakeNode}(C[i])$

PQInsert(PQ, X)

FOR $i \leftarrow 1$ to $N-1$ DO

$X \leftarrow \text{ExtractMin}(PQ)$

$Y \leftarrow \text{ExtractMin}(PQ)$

$Z \leftarrow \text{MakeNode}(X \rightarrow \text{Data} + y \rightarrow \text{Data})$

$Z \rightarrow \text{Left} \leftarrow X$

$Z \rightarrow \text{Right} \leftarrow Y$

PQInsert(PQ, Z)

$X \leftarrow \text{ExtractMin}(PQ)$

RETURN X ;

END;

EXERCISES

1. Given a set of input representing the nodes of a binary tree, write a nonrecursive algorithm that must be able to output the three traversal orders.
2. Write an algorithm for checking validity of the input, i.e., the program must know if the input is disjoint, duplicated, and has a loop.
3. Two Binary Trees are similar if they are both empty or if they are both nonempty and left and right sub trees are similar. Write an algorithm to determine if two Binary Trees are similar.
4. The degree of a node is the number of children it has. Show that in any binary tree, the number of leaves are one more than the number of nodes of degree 2 ($n_0 = n_2 + 1$).
5. Write the non-recursive algorithm to traverse a tree in preorder.
6. Draw the Expression Tree for the infix Expression $(a-b) / ((c*d)+e)$.
7. Write a non-recursive algorithm to traverse a binary tree in inorder.
8. Construct a binary tree whose nodes in inorder and preorder are given as follows:
Inorder: 10, 15, 17, 18, 20, 25, 30, 35, 38, 40, 50
Preorder: 20, 15, 10, 18, 17, 30, 25, 40, 35, 38, 50
9. Given the following inorder and preorder traversal reconstruct a binary tree
Inorder sequence D, G, B, H, E, A, F, I, C
Preorder sequence A, B, D, G, E, H, C, F, I
10. Construct the binary tree for the following sequence of nodes in preorder and inorder, respectively.
Preorder: G, B, Q, A, C, K, F, P, D, E, R, H
Inorder: Q, B, K, C, F, A, G, P, E, D, H, R
11. Draw the Huffman Tree and find codes for various characters in the table given below:

Character	A	X	?	C	M	I	L
Frequency	20	7	5	10	12	25	15

12. Draw the Binary Tree from the Traversals given below:
Pre Order: s e u p n q c I j t m
In Order: p u e n s c q m j I t
13. Draw the Huffman Tree and find codes for various characters in

the table given below:

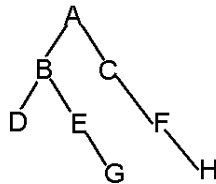
Character	A	X	?	C	M
Frequency	0.45	0.02	0.24	0.18	0.11

Decode the following msg:1011001101111100110010111101101100

14. Draw The Expression Tree for the Following:

$$-b + \sqrt{(b^2 - 4ac)} / (2a!)$$

The binary tree given below is represented in memory using array A[1:15].



15. How many array positions are vacant?

- a. 5
- b. 6
- c. 7
- d. 8

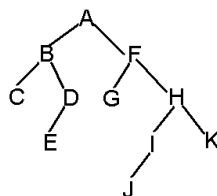
16. G is stored at array index

- a. 10
- b. 11
- c. 12
- d. None of these

17. How many array positions are vacant between F and G?

- a. 1
- b. 2
- c. 3
- d. 4

Consider the Binary tree given below



In the two-way inorder threaded representation of binary tree (without header)

18. How many right NULL pointers have been assigned with threads

- a. 5
- b. 6
- c. 7
- d. 8

19. How many left NULL pointers have been assigned with threads

- a. 4
- b. 5
- c. 6
- d. 7

20. How many nodes have been assigned with both left and threads

- a. 1
- b. 2
- c. 3
- d. 4

21. How many NULL pointers remain unassigned?

- a. 1
- b. 2
- c. 3
- d. 4

Given the characters and their corresponding frequencies.

A: 25 B:15 C:10 D:5 E:5

If Huffman tree is constructed

22. What is the depth of the tree?

- a. 2
- b. 3
- c. 4
- d. 5

23. What is weighted path length?

- a. 100
- b. 125
- c. 130
- d. 14

-
24. What is the maximum length of Huffman code for any character
- 2
 - 3
 - 4
 - 5
25. How many times merging of pair of nodes is performed?
- 2
 - 3
 - 4
 - 5
26. Binary tree is a finite set of elements. This finite set is equal to
- $\{\text{left subtree}\} \cup \{\text{right subtree}\}$
 - $\{\text{root}\} \cup \{\text{right subtree}\}$
 - $\{\text{left subtree}\} \cup \{\text{root}\}$
 - $\{\text{root}\} \cup \{\text{left subtree}\} \cup \{\text{right subtree}\}$
27. How many strictly binary tree is possible with 5 nodes?
- 1
 - 2
 - 3
 - 4

Given the traversal of a tree

Preorder Traversal: A B C D E F G H I J

Inorder Traversal: D C B E F A G I H J

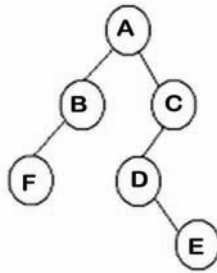
28. What is the Postorder traversal?
- D C F E B I J H G A
 - D C B E F A G I H J
 - D C B F E I J H G A
 - None of these
29. What is the root of the tree?
- A
 - D
 - J
 - None of these
30. How many elements are there in the left subtree of the root
- 5

- b. 2
 - c. 4
 - d. 5
- 36. What is the internal path length?**
- a. 9
 - b. 6
 - c. 21
 - d. 12
- 37. What is the external path length?**
- a. 9
 - b. 6
 - c. 21
 - d. 12
- 38. Which among the following is not a property of the binary tree**
- a. There is a specially designated node called the root
 - b. The rest of the nodes could be partitioned into t -disjoint sets ($t \geq 0$)
 - c. Any node should be reachable from anywhere in the tree
 - d. At most one cycle could be present in the
- 39. The maximum no of nodes in a binary tree of depth k is**
- a. $2k-1$
 - b. $2(k+1)$
 - c. $2k-1$
 - d. $2(k+1) - 1$
- 40. A binary tree with k node ($k > 1$) hasNull pointers**
- a. k
 - b. $k+1$
 - c. $k-1$
 - d. $2*k$
- 41. An inorder and postorder traversal of a binary tree was claimed to yield the**
- Following sequence
- Inorder: HAT GLOVE SOCKS SCARF GLASSES
- Postorder: GLOVE SCARF HAT GLASSES SOCKS
- a. HAT is the root of the binary tree
 - b. SOCKS is the root of the binary tree
 - c. The tree is skewed binary tree
 - d. The traversals are incorrect

- 42. For a non-empty binary tree T, if n_0 is no of nodes with degree 0 and n_2 is no of nodes with degree 2, then**
- a. $n_0 = n_2$ b. $n_0 = n_2 + 1$ c. $n_0 + 1 = n_2$ d. $n_0 < n_2$
- 43. Which among the following is not true w.r.t. binary tree and forest?**
- a. A binary tree can be empty whereas a forest can not be empty
b. Degree of any node in a binary tree is at most 2, whereas there is no upper limit on the degree of a node in forest
c. The nodes in the binary tree has the fields left, right and key whereas the forest has the fields child, sibling and father
d. The best suited data structure for representation of the Binary tree and Forest both is array.
- 44. A complete binary tree with n leaves contains**
- a. n nodes
b. $2n-1$ nodes
c. $2n$ nodes
d. $\log n$ nodes
- 45. Find a suitable match for Binary tree with naming Root**
- a. Left subtree and right subtree
b. Left descendent and right descendent
c. Left child and right child
d. Left successor and right successor
- 46. Find a suitable match for Binary tree with naming Parent**
- a. Left subtree and right subtree
b. Left descendent and right descendent
c. Left child and right child
d. Left successor and right successor
- 47. Find a suitable match for Binary tree with naming Predecessor**
- a. Left subtree and right subtree
b. Left descendent and right descendent
c. Left child and right child
d. Left successor and right successor
- 48. Find a suitable match for Binary tree with naming Ancestor**
- a. Left subtree and right subtree
b. Left descendent and right descendent
c. Left child and right child
d. Left successor and right successor

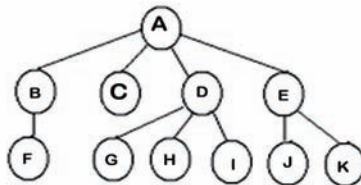
If the nodes of a binary tree is represented by the structure given below

Father Left Right Information



50. Total no. of null pointers for the father field in the tree given above is
- 1
 - 2
 - 0
 - 3
51. Total no. of null pointers for the left field in the tree given above is
- 1
 - 2
 - 0
 - 3
52. Total no. of null pointers for the right field in the tree given above is
- 1
 - 2
 - 4
 - 3

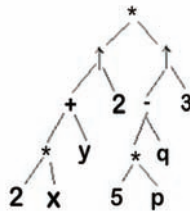
Consider the Forest given below



53. Inorder traversal of the binary tree equivalent of the forest is
- ABFC DGHIEJK
 - FBCGHIDJKEA

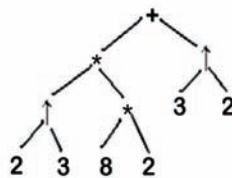
- c. F I H G K J E D C B A
 d. None of these
54. **Preorder traversal of the binary tree equivalent of the forest is**
 a. A B F C D G H I E J K
 b. F B C G H I D J K E A
 c. F I H G K J E D C B A
 d. None of these
55. **Postorder traversal of the binary tree equivalent of the forest is**
 a. A B F C D G H I E J K
 b. F B C G H I D J K E A
 c. F I H G K J E D C B A
 d. None of these
56. **What is the postorder traversal of the tree with following traversals**
 Pretraversal : A B F C D E
 Intraversal : F B A D E C
 a. F B E D C A
 b. E D C F B A
 c. C E D A B F
 d. None of these

Consider the binary tree given below



57. **The prefix form of the expression is**
 a. $* \uparrow + * 2 x y 2 \uparrow - * 5 p q 3$
 b. $2 * x + y \uparrow 2 * 5 * p - q \uparrow 3$
 c. $2 x * y + 2 \uparrow 5 p * q - 3 \uparrow *$
 d. None of these
58. **The postfix form of the expression is**
 a. $* \uparrow + * 2 x y 2 \uparrow - * 5 p q 3$
 b. $2 * x + y \uparrow 2 * 5 * p - q \uparrow 3$
 c. $2 x * y + 2 \uparrow 5 p * q - 3 \uparrow *$

- d. None of the these
59. The tree represents the expression
- $(2x + y)^2 (5p - q)^3$
 - $(2x + y)^3 (5p - q)^3$
 - $(2x + y)^3 (5p - q)^3$
 - $(2x + y)^2 (5p - q)^2$
60. Construction of a Huffman tree requires the data structures
- Binary tree & Linear Queue
 - Binary tree & Circular queue
 - Binary tree & Priority queue
 - Forest & priority queue
61. Implicit binary tree representation suits
- Strictly binary tree
 - Extended binary tree
 - Binary search tree
 - Complete binary tree
62. A complete 3-ary tree with levels <0 to n has no of elements
- $(3n+1-1)/2$
 - $(3n+1+1)/2$
 - $(3n-1)/2$
 - $(3n+1)/2$
63. What is the evaluated value of the expression given below



- 128
 - 137
 - 9
 - None of these
64. Which among the following is not an application of Tree data structure.
- Decision tree

- b. Game Programming
 - c. Arithmetic expression
 - d. Addition of Very long numbers
- 65. Which among the following is not an application of Tree data structure.**
- a. Searching
 - b. Representation of file structure
 - c. Parenthesis check for a mathematical structure
 - d. Coding for compression of text files
- 66. Which of the following pair of traversals will result in a unique binary tree**
- a. PRE – POST
 - b. PRE – IN
 - c. IN – POST
 - d. None of these

15

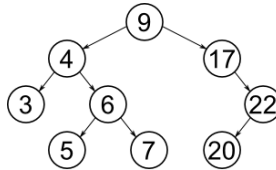
Binary Search Tree

CONTENTS

15.1. Binary Search Tree (Bst) Basics	372
15.2. Primitive Operations On Binary Search Tree (Bst).....	372
15.3. Height Balanced Search Tree: Avl Tree	379
Exercises.....	385

15.1. BINARY SEARCH TREE (BST) BASICS

Binary Search Tree (BST) is used to facilitate efficient Search. In a BST, the key value of the left subtree nodes is less than the parent's key and key value of right subtree nodes is more than the parent's key. Below given Tree is an example BST.



The InOrder Traversal of the Binary Tree results the keys in the Ascending Sequence. Usually, the BSTs are implemented using linked List with 4 fields in a Node.

Father	Left	Data	Right
--------	------	------	-------

If Deletions are not performed on the BST, the Father field can be omitted and only three field (data, left & right) is sufficient for carrying out operations.

15.2. PRIMITIVE OPERATIONS ON BINARY SEARCH TREE (BST)

- **Finding Node with Minimum Key:** By traversing the extreme left element, Node with minimum key can be found

ALGORITHM BST Minimum(Tree)

BEGIN:

P = Tree

WHILE P → Left != NULL Do

P = P → Left

RETURN P

END;

Time Complexity: ($\log_2 N$), $O(N)$

If the tree is balanced, the extreme left element will be a leaf and at a distance of $\log_2 N$ from height. In case the tree is skewed on left side, the tree height may be N.

Space Complexity: $\Theta(1)$

Only one variable P is used.

- Finding Node with Maximum key

By traversing the extreme right element, Node with maximum key can be found

ALGORITHM BST Maximum(Tree)

BEGIN:

P = Tree

WHILE P → Right ≠ NULL Do

P = P → Right

RETURN P

END;

Time Complexity: ($\log_2 N$), $O(N)$)

If the tree is balanced, the extreme left element will be a leaf and at a distance of $\log_2 N$ from height. In case the tree is skewed on left side, the tree height may be N.

Space Complexity: $\Theta(1)$

Only one variable P is used.

1. Performing Search

- Key at root node is compared with the search key. In case there is match, address to the node is returned. If there is no match, search is performed on left subtree in case search key is less than the key of current node. Search is performed on right subtree otherwise.
- The Search being performed in BST is the Binary Search.
- In case the Valid address of a node is returned through the search, The search is successful. If NULL is returned, the key being searched is not present in the BST (unsuccessful search).

ALGORITHM BSTSearch(Tree, key)

BEGIN:

P = Tree

WHILE P ≠ NULL

IF key == P → Data THEN

RETURN P

ELSE

IF key < P → Data THEN

```

P = P → Left
ELSE
P = P → Right
    RETURN NULL
END;

```

Time Complexity:

- **If Tree is Balanced:** Best Case appears when the key is at root node. Worst case when key is at the leaf node at a distance of $\log_2 N$ from root.
- **If Tree is skewed:** Best Case appears when the key is at root node. Worst case when key is at the leaf node at a distance of N from root.

(1), $\Theta(\log_2 N)$, $O(N)$

Space Complexity: $\Theta(1)$

Only 1 variable is used in the logic

- **Insertion:** The insertions in the BST always takes place as a leaf node. The appropriate position for the insertion is being searched by traversing the BST by moving Left or Right from the root Node depending on if the key to be inserted is Less than or greater than the current node's key respectively. All the insertions take place starting from root node.

ALGORITHM BSTInsert(Tree, key)

BEGIN:

```

P = Tree
Q = Null
R = MakeNode(key)
WHILE P != NULL Do
IF key < P → Data THEN
Q = P
P = P → Left
ELSE
Q = P
P = P → Right
IF key < Q → Data THEN
Q → Left = R
R → Father = Q
ELSE

```

Q→Right = R

R→Father = Q

END;

Time Complexity: (1), $\Theta(\log_2 N)$, $O(N)$

If the tree is balanced, the distance of leaf from root is $\log_2 N$. If the tree is skewed, the insertion may take place just next to Root in reverse direction of skewness. Constant effort might be required in this case. In worst case the insertion might take place at the bottom of skewed tree requiring N effort.

Space Complexity: $\Theta(1)$

P, Q, R pointer variable and memory for a node

- Recursive Insertion Process

ALGORITHM BSTInsert(Tree, key)

BEGIN:

IF Tree == NULL THEN

Tree = MakeNode(key)

ELSE

IF key < P→Data THEN

Tree→Left = BSTInsert (Tree→Left, key)

ELSE

Tree→Right = BSTInsert (Tree→Right, key)

RETURN Tree

END;

Time Complexity: (1), $\Theta(\log_2 N)$, $O(N)$

If the tree is balanced, the distance of leaf from root is $\log_2 N$. If the tree is skewed, the insertion may take place just next to Root in reverse direction of skewness. Constant effort might be required in this case. In worst case the insertion might take place at the bottom of skewed tree requiring N effort.

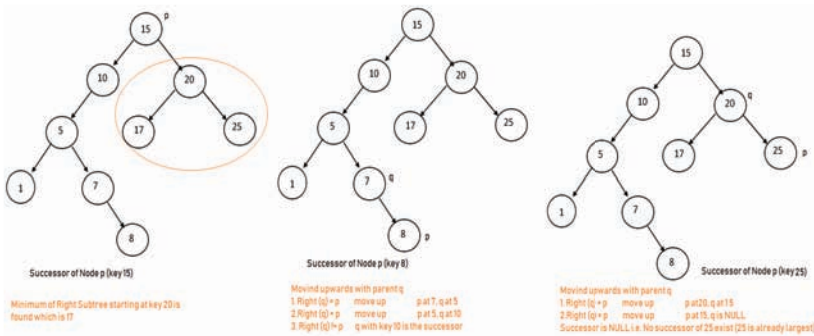
Space Complexity: (1), $\Theta(\log_2 N)$, $O(N)$

If the tree is balanced, the distance of leaf from root is $\log_2 N$ and these many activation records would be pending in stack. If the tree is skewed, the insertion may take place just next to Root in reverse direction of skewness requiring only

one activation record on stack. Constant effort might be required in this case. In worst case the insertion might take place at the bottom of skewed tree requiring N activation record pending at any given point of time.

- In Order Successor of a Node:** In order successor of a node means the node with immediate higher key. The higher values can be found on the right subtree (If Right subtree Exists). The Minimum of the Right Subtree is the successor in this case.

In case right subtree does not exist, the parent is explored for the successor. The parent for which the current node is left child, is the successor. If not then the upward movement continues until the prior condition is true or parent becomes NULL. In later case there is no successor of the given node.



ALGORITHM BSTSuccessor(P)

BEGIN:

IF $p \rightarrow \text{Right} \neq \text{NULL}$ THEN

$q = \text{BSTMinimum}(p \rightarrow \text{Right})$

RETURN q

ELSE

$q = p \rightarrow \text{Father}$

WHILE $q \neq \text{NULL}$ AND $q \rightarrow \text{Right} = p$ DO

$p = q$

$q = \text{Father}(q)$

RETURN q

END;

Time Complexity: ($\log 2N$), $O(N)$

Space Complexity: $\Theta(1)$

- In Order Predecessor of a Node:** Predecessor is the node which

appears immediately before the given node in the Inorder sequencing.

ALGORITHM BSTPredecessor(P)

BEGIN:

IF $p \rightarrow \text{Left} \neq \text{NULL}$ THEN

$q = \text{BSTMaximum}(p \rightarrow \text{Left})$

RETURN q

ELSE

$q = p \rightarrow \text{Father}$

WHILE $q \neq \text{NULL}$ AND $q \rightarrow \text{Left} = p$ DO

$p = q$

$q = \text{Father}(q)$

RETURN q

END;

Time Complexity: ($\log 2N$), $O(N)$)

Space Complexity: $\Theta(1)$

- **Deletions:** For Deletion of the node, there are three cases

a. Case 1: The target node has no child.

b. Case 2: Target node has 1 child.

c. Case 3: Target node has 2 child.

For dealing with deletions, Let us design two algorithms:

- $\text{IsLeft}(p)$ that finds if the given node is left child of its parent; and
- $\text{IsRight}(p)$ that finds if the given node is right child of its parent.

Both of these are boolean value function that either returns true or false.

ALGORITHM IsLeft(p)

BEGIN:

IF $p \rightarrow \text{Father} = \text{NULL}$ THEN

RETURN FALSE

ELSE

IF $p \rightarrow \text{Father} \rightarrow \text{Left} = p$

RETURN TRUE

ELSE

RETURN FALSE

END;

Time Complexity: $\Theta(1)$

Space Complexity: $\Theta(1)$

ALGORITHM IsRight(p)

BEGIN:

```

    IF p → Fahter == NULL THEN
        RETURN FALSE
    ELSE
        IF p → Fahter → Right == p
            RETURN TRUE
        ELSE
            RETURN FALSE

```

END;

Time Complexity: $\Theta(1)$

Space Complexity: $\Theta(1)$

ALGORITHM BSTDelete(p)

BEGIN:

Case 1: If Node to be Deleted is has 0 Child

```

IF IsLeft(p) THEN
    p → Father → Left = NULL
ELSE
    p → Father → Right = NULL
x = p → Data
FreeNode(p)
RETURN x

```

Case 2: If Node to be Deleted is has 1 Child

If p → Left == NULL AND p → Right != NULL

```
    q=p→Right
If p→Left !=NULL AND p→Right == NULL
    q=p→Left
r=p→Father
IF IsLeft(p) THEN
r→Left=q
q→Father=r
ELSE
    r→Right=q
q→Father=r

x=p→Data
FreeNode(p)
RETURN x
```

Case 3: If Node to be Deleted is has 2 Child

For this case, BST Successor is found and Deleted (Deletion of successor is from case 1 or 2). Data value of Deleted successor is kept with the node p, which had to be deleted originally.

```
q=BSTSuccessor(p)
x=BSTDelete(q) //Recursive Deletion by Case 1 or 2, whichever applicable
p→Data=x
END;
```

15.3. HEIGHT BALANCED SEARCH TREE: AVL TREE

The BST's height is dependent on the sequence of the keys to be inserted. It may get skewed resulting in $O(n)$ complexity of most of the operations. To balance it out and restrict the height in order of $\log_2 n$, AVL Tree is formed.

AVL Tree is a height balanced search tree. Balance factor of any node can only be in the range of +1 and -1. In case the Balance factor of any node in the tree becomes +2 or -2 while insertion, the rotations are performed to balance it out.

Balance Factor of a node:

Balance factor of a node (p) = Height(p→Left) – Height(p→Right)

Rotations in AVL Tree

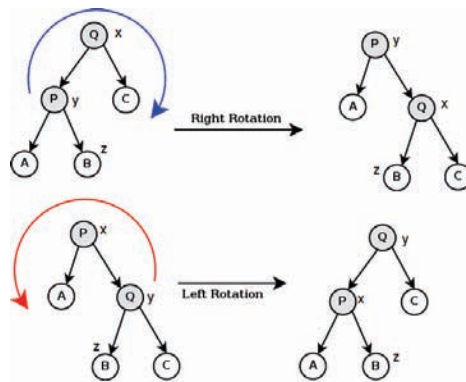
There are four rotations defined for AVL Tree

LL, RR, LR, RL

Out of this, LL, and RR are the Single rotations and LR and RL are the Double Rotations.

- **LL Rotation:** Performed when the insertion took place towards the Left of Left of the unbalanced node.
- **RR Rotation:** Performed when the insertion took place towards the Right of Right of the unbalanced node.
- **LR Rotation:** Performed when the insertion took place towards the Left of Right of the unbalanced node.
- **RL Rotation:** Performed when the insertion took place towards the Right of Left of the unbalanced node.

To make above Rotations simpler, two basic rotations are written rotateright, rotateleft



ALGORITHM RotateRight (x)

BEGIN:

```

y=x->Left
z=y->Right
y->Right =z
x->Left =z
RETURN y

```

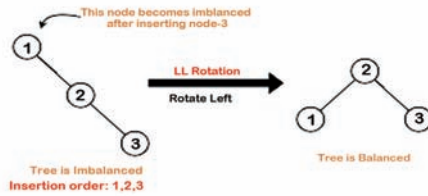
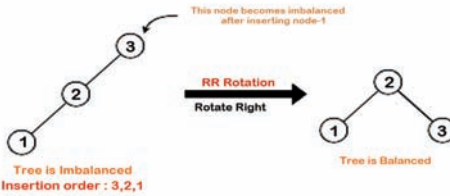
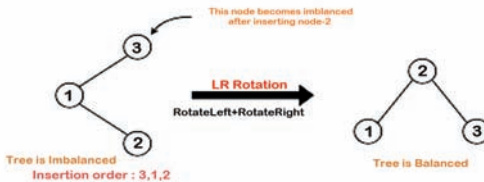
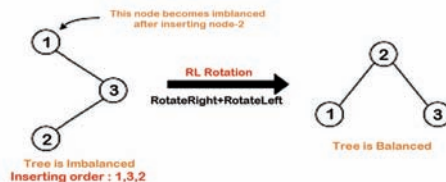
END;

Time Complexity: $\Theta(1)$

Space Complexity: $\Theta(1)$

ALGORITHM RotateLeft(x)**BEGIN:** $y = x \rightarrow \text{Right}$ $z = y \rightarrow \text{Left}$ $y \rightarrow \text{Left} = z$ $x \rightarrow \text{Right} = z$

RETURN y

END;**Time Complexity:** $\Theta(1)$ **Space Complexity:** $\Theta(1)$ **Case-02:****Case-03:****Case-04:**

ALGORITHM LL(T)**BEGIN:**

T=RotateRight(T)

RETURN T

END;**Time Complexity:** $\theta(1)$ **Space Complexity:** $\theta(1)$ **ALGORITHM RR(T)****BEGIN:**

T=RotateLeft(T)

RETURN T

END;**Time Complexity:** $\theta(1)$ **Space Complexity:** $\theta(1)$ **ALGORITHM LR(T)****BEGIN:**x= T $\rightarrow$ Left

x=RotateLeft(x)

T $\rightarrow$ Left=x

T=RotateRight(T)

RETURN T

END;**Time Complexity:** $\theta(1)$ **Space Complexity:** $\theta(1)$ **ALGORITHM RL(T)****BEGIN:**x= T $\rightarrow$ Right

x=RotateRight(x)

$T \rightarrow \text{Right} = x$

$T = \text{RotateLeft}(T)$

RETURN T

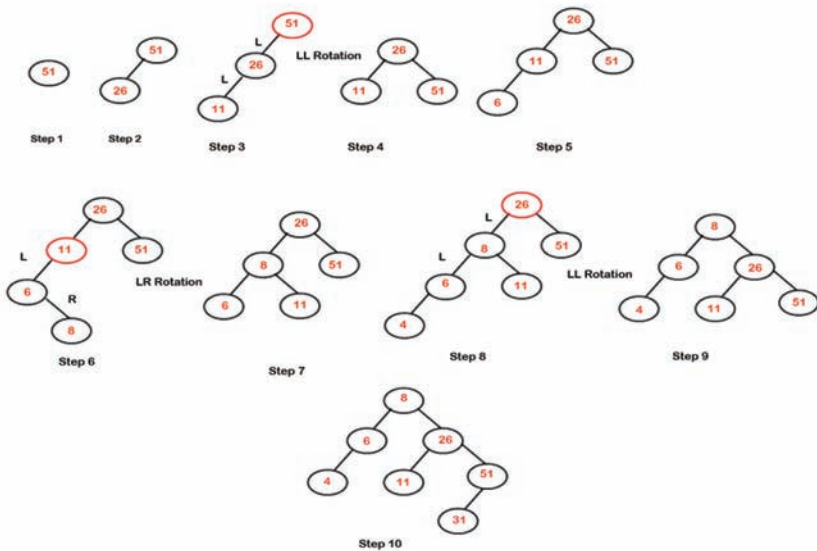
END;

Time Complexity: $\Theta(1)$

Space Complexity: $\Theta(1)$

- **Insertion in AVL Tree:** Insertions in AVL tree is done same as in BST. upon insertion, the balance factor of nodes in the insertion path is recomputed. In case the balance factor is +2 or -2, that node is rotated to get the tree balanced.

In case there are two or more unbalanced nodes on the insertion path, the nearest unbalanced node w.r.t inserted node is rotated.



ALGORITHM AVLInsertion(N, Key)

BEGIN:

IF N == NULL

N = MakeNode(Key)

h(N) = 0

ELSE

```
IF Key<N→DataTHEN
N→Left = AVLInsertion(N→Left, Key)
IF Balance(N) == 2
IF Key<N→Left→Data
N = LL(N)
ELSE
N = LR(N)
ELSE
N→Right = AVLInsertion(N→Right, Key)
IF Balance(N) == - 2
IF Key > N→Left→Data
N = RR(N)
ELSE
N = RL(N)

h(N)=Height(N)
RETURN N
END;
```

Time Complexity: ($\log 2N$), $O(N)$)

Space Complexity: $\theta(1)$

EXERCISES

1. Define the Binary Search Tree (BST). Construct a BST by insertion of following keys in the sequence 45, 36, 76, 23, 89, 115, 98, 39, 41, 56, 69, 48. Find the traversal of the constructed tree in Pre, In, and Postorder.
2. How many different BST can be constructed from the keys 1, 2 & 3?
3. Construct a AVL tree by insertion of following keys in the sequence 45, 32, 90, 34, 68, 72, 15, 24, 30, 66, 11, 50, 10. Find the traversal of the constructed tree in Pre, In, and Postorder.
4. Construct a AVL tree by insertion of following keys in the sequence. March, April, May, June, July, August, September, October, November, December, January, February. Consider dictionary order for insertion.
5. Construct a AVL tree by insertion of following keys in the sequence. 150, 155, 160, 115, 110, 140, 120, 145, 130, 147, 170, 180
6. Define the Splay Tree and Interval Tree.

If the binary search tree is constructed by the insertion of following keys in order

- | | | | | | | | |
|----|---|---|---|----|----|----|----|
| 10 | 5 | 4 | 9 | 13 | 12 | 11 | 16 |
|----|---|---|---|----|----|----|----|
7. What is the depth of the node with key 9
 - a. 1
 - b. 2
 - c. 3
 - d. 4
 8. What is the depth of the constructed tree?
 - a. 3
 - b. 2
 - c. 4
 - d. None of these
 9. How many nodes are there with the balance factor equal to zero?
 - a. 3
 - b. 4
 - c. 5
 - d. 6
 10. How many nodes are there with the balance factor equal to +1?

- a. 0
 - b. 1
 - c. 2
 - d. 3
- 11. How many nodes are there with the balance factor equal to -1?**
- a. 0
 - b. 1
 - c. 2
 - d. 3
- 12. What is the inorder traversal of the tree**
- a. 10 5 4 9 13 12 11 16
 - b. 4 5 9 10 11 12 13 16
 - c. 4 9 5 11 12 16 13 10
 - d. None of these
- 13. If we perform LL Rotation on a Binary search tree,**
- a. The Inorder traversal changes whereas preorder remains same
 - b. The Inorder traversal remains same whereas Preorder traversal changes
 - c. Both Inorder and Preorder traversals change

- d. None of these
- 14. **If we perform RR Rotation on a Binary search tree,**
 - a. The Inorder traversal changes whereas preorder remains same
 - b. The Inorder traversal remains same whereas Preorder traversal changes
 - c. Both Inorder and Preorder traversals change
 - d. None of these

Given an AVL Tree

After insertion of element 20,

- 15. **How many elements are there with Balance Factor = 2 or -2 before rotation?**
 - a. 0
 - b. 1
 - c. 2
 - d. 3
- 16. **Which rotation will be performed to make this tree balance again**
 - a. LL
 - b. LR
 - c. RR
 - d. RL
- 17. **How many nodes are there with balance factor equal to 0 after rotation**
 - a. 2
 - b. 3
 - c. 4
 - d. 5
- 18. **How many nodes are there with balance factor equal to 1 after rotation**
 - a. 0
 - b. 1
 - c. 2
 - d. 3
- 19. **How many nodes are there with balance factor equal to 1 after rotation**
 - a. 0

- b. 1
 - c. 2
 - d. 3
- 20. How many elements are there at the maximum depth**
- a. 1
 - b. 2
 - c. 3
 - d. 4
- 21. In which of the following functions in Binary search tree, use of father field is a must**
- a. Insert
 - b. Delete
 - c. Inorder-successor or logical-successor
 - d. Search
- 22. What is the logical successor of node x in Binary search tree?**
- a. Maximum of left subtree of x
 - b. Minimum of left subtree of x
 - c. Maximum of right subtree of x
 - d. Minimum of right subtree of x
- 23. What is the logical predecessor of node x in Binary search tree?**
- a. Maximum of left subtree of x
 - b. Minimum of left subtree of x
 - c. Maximum of right subtree of x
 - d. Minimum of right subtree of x
- 24. Given an AVL search tree. Insertion of 45 makes how many nodes unbalanced?**
- a. 1
 - b. 2
 - c. 3
 - d. 0
- The elements A V L T R E I S O K are inserted in an empty AVL tree?
- 25. No of LL Rotation required are**
- a. 0
 - b. 1
 - c. 2

-
- d. None of these
- 26. No of RR rotations required are**
- a. 2
b. 0
c. 1
d. None of these
- 27. No of LR rotations required are**
- a. 1
b. 2
c. 0
d. None of these
- 28. No of RL Rotations required are**
- a. 0
b. 1
c. 2
d. None of these
- 29. If LE is the external path length & LI is the internal path length in an extended binary tree with n nodes, the relation between LE & LI is given by the formula**
- a. $LE = LI$
b. $LE = LI + 2n$
c. $LE = LI + n$
d. $LE + n = LI$
- 30. Given a binary search tree with n nodes. The searching of a key in the tree requires the comparisons in**
- a. $O(1)$
b. $O(n)$
c. $O(n \log n)$
d. $O(\log n)$
- 31. Insertion of the elements Z, X, D, C, B, A in an empty BST will result in**
- a. An AVL Tree
b. A left skewed tree
c. A right skewed tree
d. A heap

A Binary search tree is created by inserting nodes 15, 20, 19, 6, 8, 4, 21,

18, 17 Answer the following questions

32. What is the root node

- a. 4
- b. 21
- c. 15
- d. None of these

33. What is the height of the created tree

- a. 4
- b. 3
- c. 5
- d. 2

34. How many nodes are there in the left subtree and right subtree of the root

- a. 5, 3
- b. 4, 4
- c. 2, 6
- d. 3, 5

35. The preorder traversal is

- a. 15 6 4 8 20 19 18 17 21
- b. 21 17 18 19 20 8 4 6 15
- c. 8 4 6 17 18 19 21 20 15
- d. None of these

36. The Post order traversal is

- a. 15 6 4 8 20 19 18 17 21
- b. 21 17 18 19 20 8 4 6 15
- c. 8 4 6 17 18 19 21 20 15
- d. None of these

37. xyz(root)

```
{  
    while(left(root)!=NULL)  
        root = left(root);  
return root;  
}
```

with respect to a binary search tree, what does the function xyz return

- a. A node with maximum value

- b. A node with the minimum value
 - c. A null pointer
 - d. None of the these
- 38. The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Which one of the following is the postorder traversal sequence of the same tree?**
- a. 10, 20, 15, 23, 25, 35, 42, 39, 30
 - b. 15, 10, 25, 23, 20, 42, 35, 39, 30
 - c. 15, 20, 10, 23, 25, 42, 35, 39, 30
 - d. 15, 10, 23, 25, 20, 35, 42, 39, 30
- 39. Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined?**
- a. {10, 75, 64, 43, 60, 57, 55}
 - b. {90, 12, 68, 34, 62, 45, 55}
 - c. {9, 85, 47, 68, 43, 57, 55}
 - d. {79, 14, 72, 56, 16, 53, 55}
- 40. A Binary Search Tree (BST) stores values in the range 37 to 573. Consider the following sequence of keys.**
- I. 81, 537, 102, 439, 285, 376, 305
 - II. 52, 97, 121, 195, 242, 381, 472
 - III. 142, 248, 520, 386, 345, 270, 307
 - IV. 550, 149, 507, 395, 463, 402, 270
- Which of the following statements is TRUE?
- a. I, II and IV are inorder sequences of three different BSTs
 - b. I is a preorder sequence of some BST with 439 as the root
 - c. II is an inorder sequence of some BST where 121 is the root and 52 is a leaf
 - d. IV is a postorder sequence of some BST with 149 as the root
- 41. A binary search tree is used to locate the number 43. Which of the following probe sequences are possible and which are not?**
- a. 61 52 14 17 40 43
 - b. 2 3 50 40 60 43
 - c. 10 65 31 48 37 43
 - d. 81 61 52 14 41 43
 - e. 17 77 27 66 18 43

16

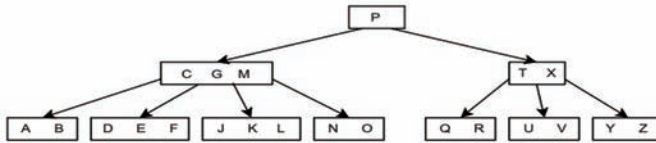
B-Tree & B+-Tree

CONTENTS

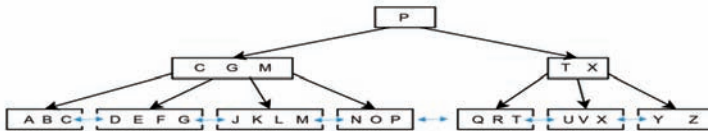
16.1. B-Tree	394
16.2. B ⁺ Tree	396
Exercises	399

B-Tree and B⁺-Tree are multi-way search Trees. Consider a Binary search tree (BST) with a single key in the node. There are two decisions which has to be taken from the current node: go to left or go to right for search. If we have 2 keys, X & Y e.g., in a node there will be 3 paths for search possible. One the path having keys less than X, another one in between X and Y, and the last one greater than Y. The search tree in which there are multiple keys allowed, are called multiway search Tree.

Two specialized multi-way search trees are: B-Tree & B⁺-Tree.



B-Tree

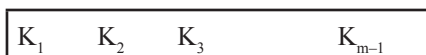


B⁺Tree

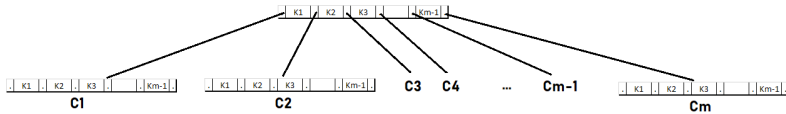
16.1. B-TREE

B-Tree is a balanced multiway search tree with the following properties

- A tree is defined by an Order (m). Order defines the maximum children a node can have.
- No of keys in an order m B-Trees
- Maximum keys m-1
- Minimum keys Floor((m-1)/2)
- The root node can violate the minimum key condition and can have 1 key also.
- No node can violate the maximum key condition
- All Leaf nodes are at the same level
- Keys in a node are arranged in nondecreasing order $K_1 \leq K_2 \leq K_3 \leq \dots \leq K_n$



- Keys in Child nodes will follow the below given relation



$$\text{Key}(C_1) \leq K_1 \leq \text{Key}(C_2) \leq K_2 \leq \dots \leq \text{Key}(C_{m-1}) \leq K_{m-1} \leq \text{Key}(C_m)$$

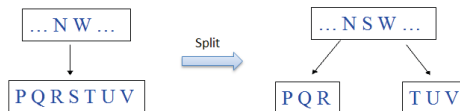
- In B-Tree, any node be it internal or leaf node can contain the key that may be searched.

16.1.1. Insertion in B-Tree

For insertion of keys in the B-Tree, the search is performed to find the appropriate node for insertions. The insertion always takes place at a leaf node. While insertion there are two possibilities

The target node contains less than m-1 keys: perform the insertion of the key at the right place in the node.

The target node contains exactly m-1 keys: In this case, virtual insertion of the key is performed. The node gets splitted into two nodes. Elements before the median keys are kept in one node. Elements after the median keys are kept in another node. The median key is sent to the parent node. In case parent node does not exist, the parent node is created and tree height increases by 1. In case the parent node exists and parent node also has m-1 keys from before, this node also splits in the same way and median key is sent upwards.



16.1.2. Order Computation in B-Tree

Consider a database table given below:

Roll No	Name	Fathers Name	Address	Contact no
98201	Abhishek Yadav	Pankaj Yadav	C-5, Alambagh, Lucknow	05222489987
98202	Adesh Kumar Singh	Katar Singh	346, Patel Nagar 2, Ghaziabad	01207135487
98203	Akhilesh Kr Srivastava	Ramesh Srivastava	C-13/111, Lahangpura, Varanasi	05422411999
...				

The table gets stored in the permanent storage and search may be performed on this table anytime. Usually, search is around the key value (In this table it is the Roll Number). Expected result is information related to Roll No (e.g., 98203.) and

information required is record related to Roll No 98203. A B-Tree stores the key value & Record addresses pair of all the records in a table. Whenever a search is performed for a particular key, the record address for that key can be the output of the B-Tree.

B-Tree are stored on a permanent storage. Usually, a B-Tree node fits exactly in the Hard Disk Sector. Each sector is of 512 Bytes hence the size of each B-Tree node is also 512 Byte or less.

Structure of B-Tree Node

C_1	$\langle K_1, R_1 \rangle$	C_2	$\langle K_2, R_2 \rangle$	C_3	$\langle K_3, R_3 \rangle$	...	$\langle K_{m-2}, R_{m-2} \rangle$	C_{m-1}	$\langle K_{m-1}, R_{m-1} \rangle$	C_m
-------	----------------------------	-------	----------------------------	-------	----------------------------	-----	------------------------------------	-----------	------------------------------------	-------

There are:

m-1 keys

m-1 record address

m Child node pointers

If keys, Record Addresses, child node pointer requires K, R, and C bytes for storage then

$$(m-1)*K + (m-1)*R + m*C \leq 512$$

Example:

If K=8 Bytes, R=12 Bytes, C=2 Bytes

$$(m-1) 8 + (m-1) 12 + m*2 \leq 512$$

$$22m - 20 \leq 512$$

$$22m \leq 532$$

$$m \leq 532/22$$

$$m \leq 24.1818$$

So the order of a B-Tree is either 24 or less

16.2. B⁺TREE

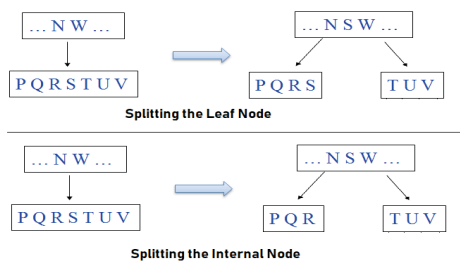
B⁺Tree is similar to B-Tree except that the keys are stored only at the leaf nodes (In B-Tree the keys can be there in any node, be it leaf or internal). Internal nodes contains the indexes that facilitate to reach to the Leaf node. All the leaf nodes are connected to each other. In the Database, these tables are organized into a B-Tree for searches. In case the entire table is required to be searched, the first key is searched from the root node of the B⁺Tree to reach to the leaf containing key. As all the leaf nodes are connected, remaining keys can be found by traversing the leaf nodes without starting search from root node.

16.2.1. Insertion in the B⁺Tree

For insertion of keys in the B⁺Tree, the search is performed from root node to find the appropriate leaf node for insetsions. The insertion is always at a leaf node. While insertion there are two possibilities

The target node contains less than m–1 keys: – perform the insertion of the key at the right place in the node.

The target node contains exactly m–1 keys:- In this case, virtual insertion of the key is performed. The node gets splitted into two nodes. Elements upto median keys are kept in one node. Elements after the median keys are kept in another node. The median key is sent to the parent node (Thereby the median key is maintained on the leaf node and copy of this is sent upwards as key in the index nodes). In case parent node does not exist, the parent node is created and tree height increases by 1. In case the parent node exists and parent node also has m–1 from before, one extra key in this node will violate the maximum condition hence this node also splits but while splitting this node, median key is not retained (as this is an index node). Median key is sent upwards in the parent node.



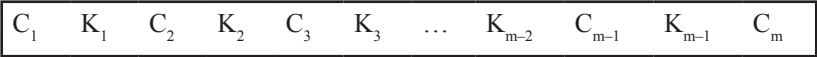
16.2.2. Order Computation in B⁺ Tree

In B+-Tree, The structure of the internal node is different from that of the Leaf node. Internal node only contains the indexes and address of the child nodes. Leaf nodes contains key, Address of record, and pointer to the sibling nodes.

B⁺Tree are stored on a permanent storage. Usually, a B⁺Tree node fits exactly in the Hard Disk Sector. Each sector is of 512 Bytes hence the size of each B⁺Tree node is also 512 Byte.

- Structure of B⁺Tree Node

Internal Node



There are

m-1 keys

m Child node pointers

If keys, Record Addresses, child node pointer requires K, R, and C bytes for storage then

$$(m-1)*K+m*C \leq 512$$

Leaf Node

$P_{\text{Left}} \quad \langle K_1, R_1 \rangle \quad \langle K_2, R_2 \rangle \quad \langle K_3, R_3 \rangle \quad \dots \quad \langle K_{m-2}, R_{m-2} \rangle \quad \langle K_{m-1}, R_{m-1} \rangle \quad P_{\text{Right}}$

There are

m-1 keys

m-1 record address

2 pointers to left and right siblings (in some cases it is only for the right sibling)

If keys, Record Addresses, Sibling node pointer requires K, R, and S bytes for storage then

$$(m-1)*K+(m-1)*R+2*S \leq 512 \text{ (If left and right both siblings are recorded)}$$

$$(m-1)*K+(m-1)*R+1*S \leq 512 \text{ (If only right siblings is recorded in the leaf node)}$$

Example:

If K=8 Bytes, R=12 Bytes, C/S=2 Bytes

Order of Leaf Node

$$(m-1) 8 + (m-1) 12 + 2*2 \leq 512$$

$$20 m - 16 \leq 512$$

$$20 m \leq 528$$

$$m \leq 528/20$$

$$m \leq 26.4$$

So the order of a B+Tree Leaf node is either 26 or less

Order of Internal Node

$$(m-1)K+mC \leq 512$$

$$(m-1)*8+m*2 \leq 512$$

$$10m \leq 520$$

$$m \leq 52$$

So the order of a B+Tree internal nodes is either 52 or less

EXERCISES

- 1) Construct a B-Tree of order 3 for the following set of Input data: 69, 19, 43, 16, 25, 40, 132, 100, 145, 7, 15, 18.
- 2) Show the result of inserting the keys. F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E in the order to an empty B-tree of Order 5.
- 3) Show the result of inserting the keys. F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E in the order to an empty B⁺tree of Order 5.
- 4) Find the maximum no of keys possible in a B-Tree of Height h
- 5) Find the minimum no of keys possible in a B-Tree of Height h
- 6) Draw the B-Tree of Order 5 by insertion of following keys in the sequence:

A G F B K D H M J E S I R X C L N T U P

17

Graph

CONTENTS

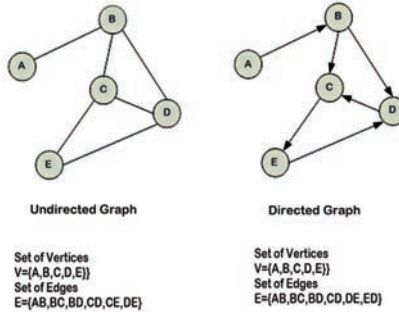
17.1. Graph Basics	402
17.2. Applications Of Graphs	405
17.3. Representing Graphs.....	406
17.4. Graph Traversal.....	407
17.5. Shortest Path	412
17.6. Transitive Closure.....	415
17.7. Spanning Tree	418
17.8. Topological Sort	421
Exercises	424

17.1. GRAPH BASICS

The graph is a Non-Linear Data Structure. It is the collection of **vertices** and **edges**. In a graph, a node can have any number of connections to vertices via edges.

17.1.1. Graph Terminology

Graphs can be directed or undirected. The directions are specified on the edges in the directed graphs. There is no direction on the edges in the undirected graphs.



Degree of a Node: Degree of a node is the edges containing that node. In the given graph

Degree(A): -1

Degree(B): -3

Degree(C): -3

Degree(D): -3

Degree(E): -2

In Degree: In Degree of a node is the edges terminating at a node.

Out Degree: Out Degree of a node is the edges emerging from a node.

InDegree(A): -0

InDegree(B): -1

InDegree(C): -2

InDegree(D): -2

InDegree(E): -1

OutDegree(A): -1

OutDegree(B): -2

OutDegree(C): -1

OutDegree(D): -1

OutDegree(E): -1

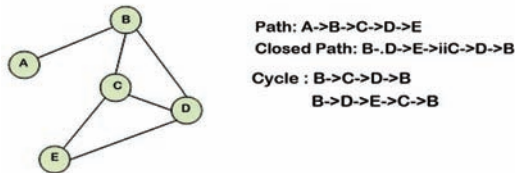
In the directed graph, if there is an edge from A to B as in the figure,

- The vertices A and B are adjacent (neighbors).
- Node A is the predecessor of node B.
- Node B is the successor of node A.

In the undirected graph, if there is an edge between node A to B then:

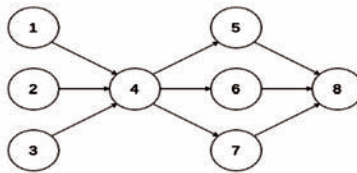
- Nodes A and B are adjacent (they are neighbors).
- **Path:** A path is a walk from one node to the other. The first vertex is called the source and last one is called Destination. No of edges falling in a path is called Length of the path.
- **Simple Path:** A simple path is a path in which no vertex is repeated (except source and destination, which may be same or different)
- **Closed path:** A path with same source and destination is called a closed path
- **Cycle:** A simple closed path of length 3 or more is called a cycle

Consider the graph given in figure:

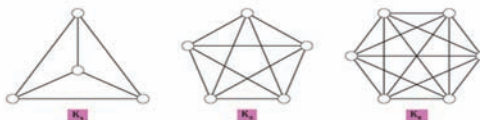


17.1.2. Some Special Types of Graphs

- A directed graph that has **no** cyclic paths (that contains no cycles) is called a **DAG** (a Directed Acyclic Graph).



- An undirected graph that has an edge between every pair of vertices is called a complete graph.



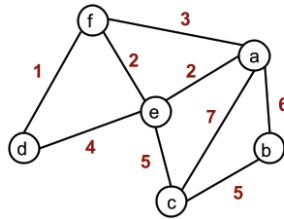
Total No of edges in an undirected complete Graph is $n(n-1)/2$ where n represents the no of vertices.

A directed Graph having an edge from each vertex to the other vertices is also a complete Graph. The total number of edges in this case will be $n(n-1)$.

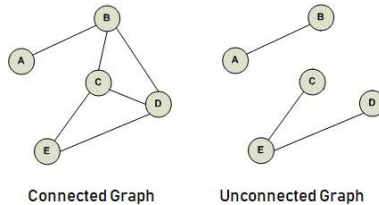
- If weights are associated with each edge in the Graph, the Graph is called a weighted graph.

These weights can represent:

- Cost for traveling from one endpoint of the edge to the other.
- Distance between endpoints of the edge.
- Time for traveling from one endpoint of the edge to the other.

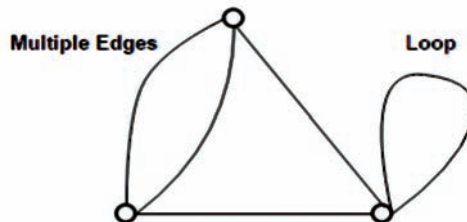


- A Graph is connected if, treating all edges as being undirected, there is a path from every vertex to every other vertex. For example:



17.1.2.1. Multi Graph

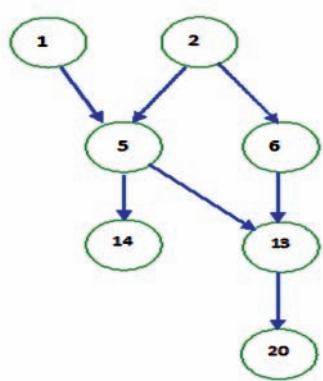
A graph having parallel/multiple edges (edges with the same source and destination) and self-loops (same endpoints) is called the Multi Graph



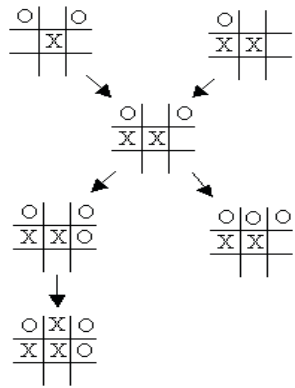
17.2. APPLICATIONS OF GRAPHS

The vertices of a graph represent objects, and the edges represent relationships. Consider some scenarios as given under

- **Trains between a Pair of Stations:** The vertices represent the stations, and the edges represent the train connectivity. The weights assigned on the edges may represent the distance/time/cost of travel between the stations.
- **Dependent Tasks:** The vertices represent the tasks. The edges denote the completion of the tasks. If there is an edge $a \rightarrow b$, means a happens before b or for b to start, completion of a is a must.



- **Flow Charts/Control-Flow Graphs:** The vertices denote the elements of the Logic and edges denote the flow of the statements in the logic.
- **State Transition Diagrams:** The vertices denote the states, edges denote the moves from one state to another. e.g., in a Tic Tac Toe game, the possible moves are represented in the form of a state transition diagram.



The graph is a standard way to represent the scenarios mentioned above, and there are standard Algorithms to answer certain questions like:

- What is the shortest path from Delhi to Varanasi by train or which route takes least time for travel from Delhi to Varanasi?
- What are the jobs required to be done before doing 20 in a factory?
- To execute some statement no i in the logic, what are the other statements which will precede?
- Starting from the current state, In how many steps I can win the game of Tic-tac-toe?

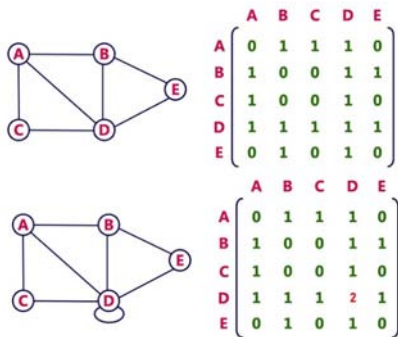
17.3. REPRESENTING GRAPHS

A graph can be represented in three ways in the memory:

- Adjacency matrix;
- Incidence matrix; and
- Adjacency list.

17.3.1. Adjacency Matrix

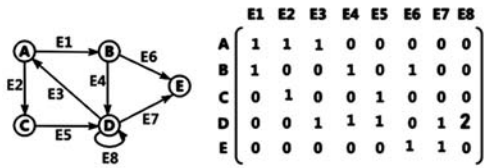
It is a $N \times N$ square Matrix where N is the no of vertices. The connections are represented by 1 and no connections are represented by 0. For Self loops connection is shown by entry 2 in the adjacency matrix.



Sum of all the entries in the adjacency matrix is $2 \times$ number of edges in case of undirected graph and same as the number of edges in directed graph.

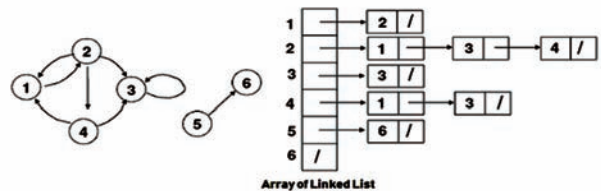
17.3.2. Incidence Matrix

In an incidence matrix row represents the edges and columns the edges. The endpoints of the edges are represented by value 1 in case of normal edges and 2 in case of self-loops. The column wise sum for each column is 2.



17.3.3. Adjacency List

The adjacency list is the array of linked list. No of elements in the array is same as the number of vertices. With each linked list, connections are shown by adding a node in sequence, in the form of linear linked list nodes.



17.3.3.1. Accessing Nodes by Name

If the names are assigned to the vertices, e.g., Varanasi, a fast look up table, e.g., Hash Table is required to map the names with node number. This is because most of the Graph algorithms assume the node numbers as a base for computations.

17.4. GRAPH TRAVERSAL

The fundamental problems in the Graph is Traversal. There are two standard methods of traversing the vertices/edges in a graph in a systematic way:

- Breadth-first search (BFS); and
- Depth-first search (DFS).

For traversal of the graph, there are three possibilities associated with the vertices: undiscovered, explored, completed.

In our description of node discovery, the vertices are assigned three colors:

- **Undiscovered:** Initially (WHITE in color);
- **Explored:** (GRAY in color); and
- **Completed:** The vertex after all visiting all its neighbors (BLACK in color).

To start with, a single vertex is taken and edges outgoing from this are explored.

- If the edge leads to an undiscovered vertex, it is marked explored and added to the list of discovered vertices.
- If an edge leads to a vertex which has already been explored or completed, the vertex is ignored.

17.4.1. Time Complexity

Each edge is visited either once (for directed graphs), or twice (in undirected graphs) hence the time complexity is $\Theta(|V| + |E|)$.

- Depending upon the way the explored vertices are stored, we obtain the two methods of traversal, the BFS, and DFS:
- **Queue:** The vertex explored first are the first one to finish. The connecting vertices to the vertex explored are kept in the queue. The vertices from the queue are deleted in FIFO manner and their neighbor vertices are inserted in the queue if they have not been explored earlier.
- **Stack:** The vertex explored first are the last one to finish. The connecting vertices to the vertex explored are kept in the stack. The newest explored vertex is looked for the neighboring vertices which are still unexplored.

17.4.2. Breadth-First Search (BFS)

- All the vertices are initialized to white in color.
- An empty queue is defined that is used to keep the grey colored vertices.
- The source vertex s is inserted in the queue and its color is defined to grey.
- A distance vector $d[]$ is maintained which defines the length of the path from source s to a given vertex.
- Distance of source, i.e., $d[s]$ initialized to 0.
- Predecessor of each vertex is maintained in $\Pi[]$. If a there is an edge (u, v) and v is explored from u in the BFS, the predecessor of v is set as u .
- The vertex deleted from the Queue is marked Black in color.
- After the execution of BFS, the $d[]$ tells the minimum distance from source to given vertex.

ALGORITHM BFS (G, s)

BEGIN:

QUEUE Q

Initialize (Q)

```

For all  $u \in V[G]$  DO
   $\Pi[u] \leftarrow \text{Nil}$ 
   $\text{Color}[u] \leftarrow \text{White}$ 
  EnQueue (Q, s)
   $\text{Color}[s] \leftarrow \text{Grey}$ 
   $d[s] \leftarrow 0$ 

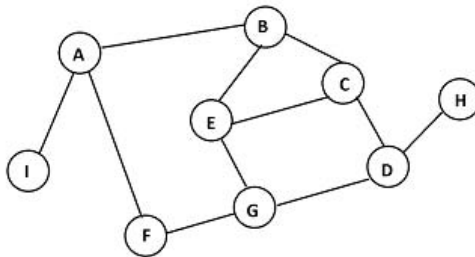
  WHILE (!Empty(Q)) Do
     $u \leftarrow \text{DeQueue}(Q)$  Do
      For each  $V \in \text{Adj}[u]$ 
        If  $\text{Color}[u] == \text{White}$ 
           $\text{Color}[u] \leftarrow \text{Grey}$ 
          EnQueue (Q, u)
           $d[v] \leftarrow d[u] + 1$ 
           $\Pi[u] \leftarrow u$ 
           $\text{Color}[u] \leftarrow \text{Black}$ 
          WRITE (u)
  END;

```

Time Complexity: $\Theta(|V| + |E|)$

Space Complexity: $\Theta(|V|)$

Consider the graph given below.



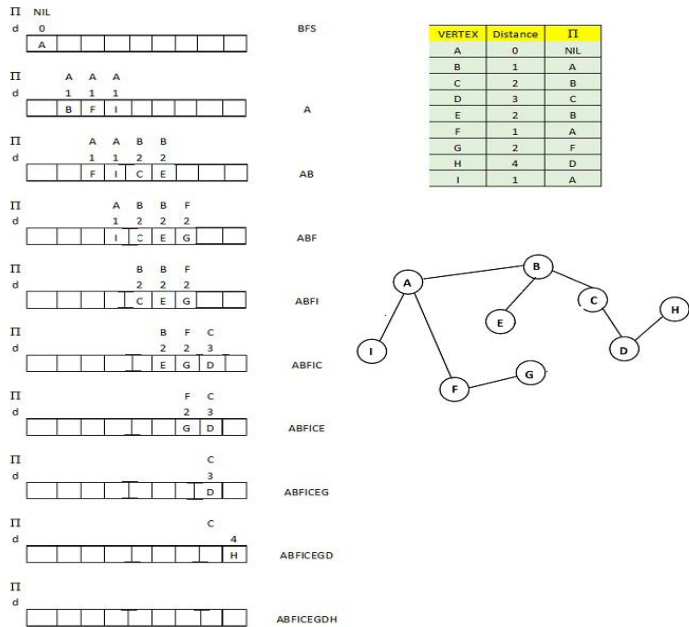
Adjacency list is given as:

A: B, F, I

B: A, C, E

C: B, D, E

D: C, G, H
E: B, C, G
F: A, G
G: D, E, F
H: D
I: A



- Start time $S[u]$ defines the time when a vertex is first visited.
- Finish time $F[u]$ defines the time at which all the neighboring vertices have been explored completely
- Predecessor of each vertex is maintained in $\Pi[]$. If there is an edge (u, v) and v is explored from u in the DFS, the predecessor of v is set as u .

ALGORITHM DFS (G)**BEGIN:**

For all $u \in V[G]$ Do

Color[u] $\leftarrow$ White

Time $\leftarrow 0$

For all $u \in V[u]$ Do

IF Color[u] $\leftarrow$ White

DFS $\leftarrow$ Visit (u)

END;**ALGORITHM DFS-VISIT (U)****BEGIN:**

Color[u] $\leftarrow$ Grey

$S[u] \leftarrow$ Time+1

For all $V \in \text{Adj}[U]$ Do

IF Color [V] $\leftarrow$ White

$\Pi[V] \leftarrow U$

DFS $\leftarrow$ Visit [V]

Color [U] $\leftarrow$ Black

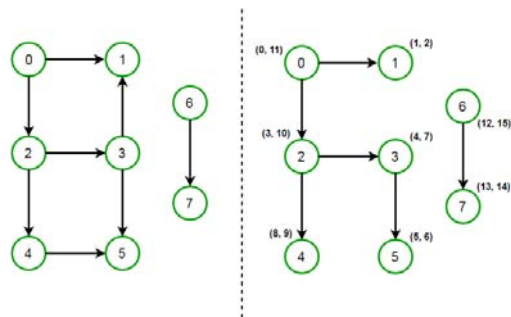
$F[U] \leftarrow$ Time+1

Time $\leftarrow$ Time+1

END;

Time Complexity: $\Theta(|V|+|E|)$

Space Complexity: $\Theta(|V|)$



In The Graph on the right side, (x, y) represent the pair of start and finish time for a vertex. Start time denotes the time at which the vertex was explored.

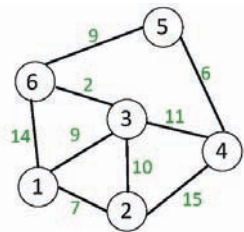
Vertex	Connections	Start Time	Finish Time	Predecessor Π
0	1, 2	0	11	NIL
1		1	2	0
2	3, 4	3	10	0
3	1, 5	4	7	2
4	5	8	9	2
5		5	6	3
6	7	12	15	NIL
7		13	14	6

DFS for the graph given above is the time at which they were first discovered i.e., 0, 1, 2, 3, 5, 4, 6, 7

The graph on right side also shows the discovery of the DFS Forest (Predecessors helped to form it).

17.5. SHORTEST PATH

If the graph represents the cities/stations and edges represent the connection between them. Shortest path is the path through which the walk is of minimal length for given source and destination. e.g., consider the graph given below.



There are multiple paths possible from 1 to 5.

$1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ length 28

$1 \rightarrow 6 \rightarrow 5$ length 23

$1 \rightarrow 3 \rightarrow 6 \rightarrow 5$ length 20

$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ length 34

Out of the available paths, $1 \rightarrow 3 \rightarrow 6 \rightarrow 5$ is the minimal path of length 20.

17.5.1. All Pairs Shortest Path

Consider a virtual scenario wherein during the election the election commission has decided to pay the travel cost to the voters residing in other parts of the country for earning their livelihood but have their names in the voter list of their home town only. This has been done to improve the overall voting percentage. Election commission has decided to pay to the voters for their travel from work city to native city by shortest route by train. To avoid any run time issues, election commission officials will find the shortest path between every pair of stations in the country.

Below given Algorithm is finding the shortest path between every pair. Weight matrix for the Graph with N vertices is given as the input. This represents the cost of edge in case that exist between a pair of vertices, and 0 otherwise. The Algorithm considers if the edge does not exist between a pair of vertex, considering end points as source and destination, they are unapproachable and their distance is set to infinite.

The Algorithm takes each of the vertex (k) as mediator one time and computes the path length. The least among direct path between source & destination ($i \rightarrow j$) or path via mediator ($i \rightarrow k \rightarrow j$) is taken as new entry in the matrix at the (i, j) cell

$W[i][j] \leftarrow \text{Min}(W[i][j], W[i][k] + W[k][j])$

ALGORITHM APSPFloydWarshall (W [] [], N)

BEGIN:

FOR i $\leftarrow$ 1 To N Do

FOR i $\leftarrow$ 1 To N DO

IF $W[i][j] = 0$

IF $i \neq j$ THEN

$W[i][j] = \infty$

FOR k $\leftarrow$ 1 To N Do

FOR i $\leftarrow$ 1 To N DO

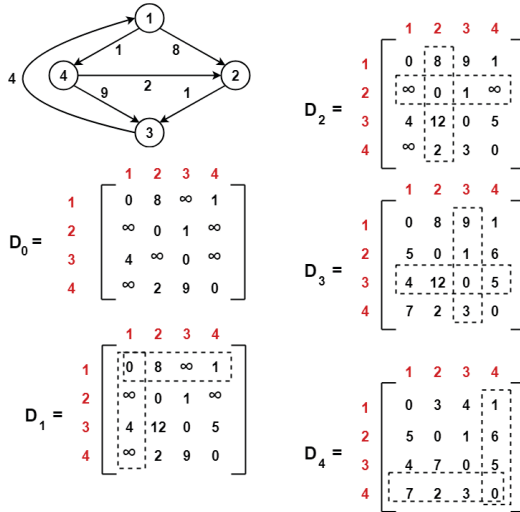
FOR j $\leftarrow$ 1 To N DO

$W[i][j] \leftarrow \text{Min}(W[i][j], W[i][k] + W[k][j])$

END ;

Time Complexity: $\Theta(N^3)$

Space Complexity: $\Theta(1)$



The Above Diagram shows the computation of Shortest path using the Floyd Warshalls Algorithm. D_0 Matrix Shows the initial distance matrix with all 0 entries (except diagonal) converted to infinite. D_1, D_2, D_3, D_4 show the updated Distance matrix considering vertex 1, 2, 3, 4 as mediator. For D_1, D_0 is the base, For D_2, D_1 is the base, For D_3, D_2 is the base, D_4, D_3 is the base. D_4 is the final Matrix with the shortest path.

17.5.2. Single-Source Shortest Path

If you are booking a cab from any app-based taxi service, you set source and destination. The app calculates the shortest path and tells you the fare in advance. This is possible only if the app computes the shortest path between the selected source and destination at the run time.

Consider a scenario in which a reputed college is organizing a national conference. All the delegates will be paid for their travel to the conference. The College needs to calculate the shortest path of all the destinations from College city in advance. The single source, in this case, is the college city.

The below-given Algorithm computes the shortest path to all the destinations considering a single source using Dijkstra Algorithm. In the algorithm a priority queue is taken that stores the vertices. The distance of source is initialized to 0 and all other vertices as ∞ . The minimum distance vertex is deleted from the queue and

connections for this are updated for distance (in case the new distance is shorter than previously computed). The predecessor is also updated. The operation continues by the time there is no vertex in the queue.

ALGORITHM SSSPDijkstra (G , W [] [] , S)

BEGIN:

Priority Queue PQ

Initialize (PQ)

For all $U \in V [G]$ DO

$D [U] \leftarrow \infty$

$\Pi[U] \leftarrow NIL$

PQ Insert (PQ , U)

$D [S] \leftarrow 0$

WHILE ! Empty (PQ) DO

$U \leftarrow \text{ExtractMin(PQ)}$

For All $V \in \text{Adj} [U]$ AND $V \in \text{PQ}$

IF $d[V] < d[U] + W [U][V]$

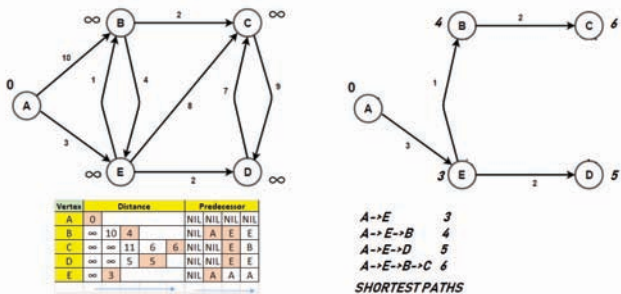
$d[V] \leftarrow d[U] + W [U][V]$

$\Pi[V] \leftarrow U$

END ;

Time Complexity: $O(|E|+|V|\text{Log } |V|)$

Space Complexity: $\Theta(|V|)$



17.6. TRANSITIVE CLOSURE

Transitive Closure of a Graph is a graph that shows the existence of path between a pair of vertex as the direct edge. There are two methods by which the transitive closure can be found.

- Warshall method; and
- Matrix multiplication method.

17.6.1. Warshall Algorithm for Transitive Closure

As in the Floyd Warshall Algorithm for All Pairs Shortest path, here too the existence of path is updated using the mediator vertices. All vertices are taken as mediator one by one. The process goes like:

- Find adjacency matrix of the graph.
- Update the adjacency matrix by taking a mediator vertex in search of a path. This is done by the formula $A[i][j] = A[i][j] \text{ OR } (A[i][k] \text{ AND } A[k][j])$.
- Either there is a direct path between $A[i][j]$ or a path via k i.e., $A[i][k]$ and $A[k][j]$. In both cases the $A[i][j]$ entry is set to 1.
- The process repeats for each vertex as a mediator for all the entries of the adjacency matrix.
- The newly created adjacency matrix taking k as mediator works as the base for computing adjacency matrix taking $k+1$ as the mediator.

ALGORITHM TransitiveClosureWarshall (A[][], N)

BEGIN:

FOR $k \leftarrow 1$ To N Do

FOR $J \leftarrow 1$ To N DO

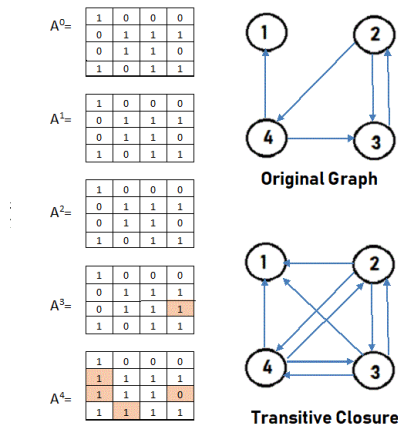
FOR $j \leftarrow 1$ To N DO

$A[i][j] \leftarrow A[i][j] \mid (A[i][k] \ \& \ A[k][j])$

END ;

Time Complexity: $\Theta(N^3)$

Space Complexity: $\Theta(1)$



17.6.2. Matrix Multiplication Method

Adjacency Matrix of the Graph shows the length 1 path.

Find $A^2 = A \times A$. This matrix represents length 2 paths

Find $A^3 = A^2 \times A$. This matrix represents length 3 paths

Find $A^4 = A^3 \times A$. This matrix represents length 4 paths

...

Find $A^N = A^{N-1} \times A$. This matrix represents length N paths

Where N is the number of vertices of the graph

Find $M = A^1 + A^2 + A^3 + \dots + A^N$

Find a Transitive Closure Matrix T by replacing all non zero entries in M by 1

ALGORITHM TransitiveClosure(A[][], N)

BEGIN:

$M[N][N] \leftarrow A$

$B[N][N] \leftarrow A$

$T[N][N] \leftarrow \{0\}$

For $i \leftarrow 2$ TO N DO

$B \leftarrow \text{MatrixMultiply}(B, A)$

$M \leftarrow M + B$

FOR $i \leftarrow 1$ TO N DO

FOR $j \leftarrow 1$ TO N DO

IF $M[i][j] \neq 0$ THEN

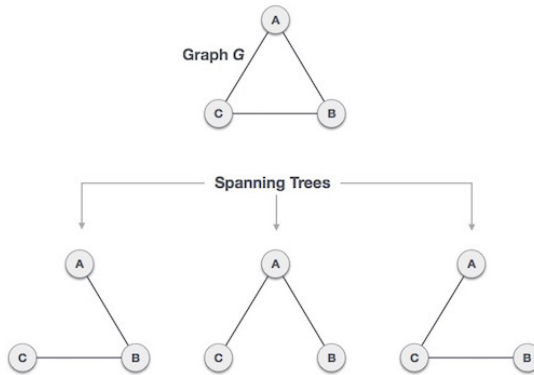
$T[i][j] \leftarrow 1$

RETURN T

END;

17.7. SPANNING TREE

Spanning Tree T of any Graph ($G=(V, E)$) is a connected acyclic Graph with same vertices set.



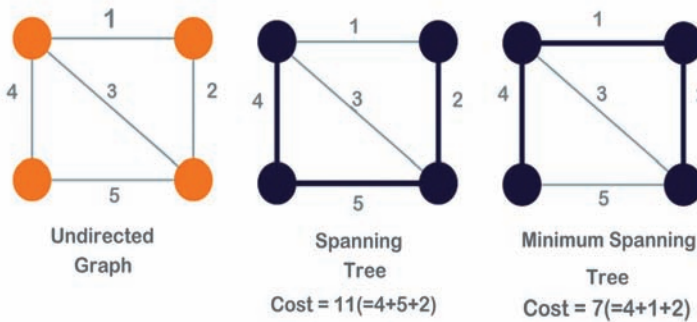
Many spanning trees are possible for a graph.

For a complete graph with N vertices, total no of spanning trees possible are N^{N-2} . e.g.,

N= 3	Total Spanning Tree $3^{3-2} = 3$
N= 4	Total Spanning Tree $4^{4-2} = 16$
N= 5	Total Spanning Tree $5^{5-2} = 125$

17.7.1. Minimal Spanning Tree

Consider a scenario of an office in which intercom connection is required to be established between various offices. The worst case scenario is to connect each office with each of the other office. The better solution would be that connections are established such that every office is connected to other by some means. In such a case less wires would be required. To find a connection such that minimum wires are required would result in the Minimal Spanning Tree.



There are two ways of finding the minimal spanning tree:

- Prim method; and
- Kruskal method.

17.7.2. Prim's Algorithm for Minimal Spanning Tree

For the run of the algorithm a Weight Matrix is given and a root node of the Spanning Tree.

The below given Algorithm computes the minimal Spanning Tree using the Prim's Algorithm. In the algorithm a priority queue is taken that stores the vertices. The key of root is initialized to 0 and all other vertices as ∞ . The minimum key vertex is deleted from the queue and connections for this are updated for key (in case the new key is shorter than previously computed). The predecessor is also updated in case of key update. The operation continues by the time there is no vertex in the queue.

ALGORITHM MST Prims ($G, W[][], R$)

BEGIN:

Priority Queue PQ

For all $U \in V[G]$ Do

Key[U] $\leftarrow \infty$

$\Pi[U] \leftarrow \text{NIL}$

PQ Insert (PQ , U)

Key [R] $\leftarrow 0$

WHILE !Empty(PQ) Do

U $\leftarrow \text{ExtarctMin (PQ)}$

For all $V \in \text{Adj [U]}$ AND $V \in \text{PQ}$ DO

IF $W[U][V] < \text{Key [V]}$ DO

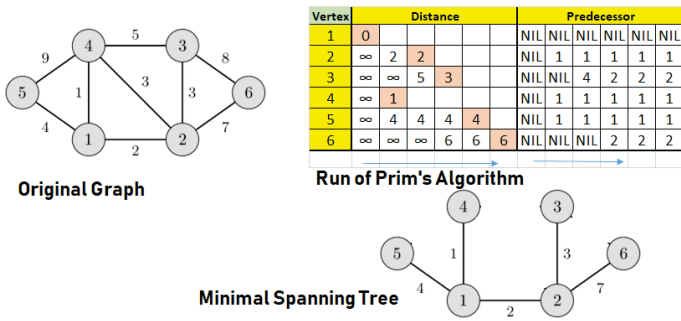
Key [V] $\leftarrow W [V][U]$

$\Pi[U] \leftarrow U$

END;

Time Complexity: $O(|E|+|V|\text{Log } |V|)$

Space Complexity: $\Theta(|V|)$



17.7.3. Kruskal's Algorithm for Minimal Spanning Tree

Kruskal Algorithm arranges the edges in the ascending sequence. The edges are selected one by one and are added to the set of selected edges in the spanning tree if the edge does not form any cycle. The detection of cycle in the spanning tree is done through a data structure named as Disjoint set data structure.

- **Disjoint Set Data Structure:** Some applications involve grouping n distinct objects into a collection of disjoint sets. Two important operations are then finding which set a given object belongs to and uniting the two sets.

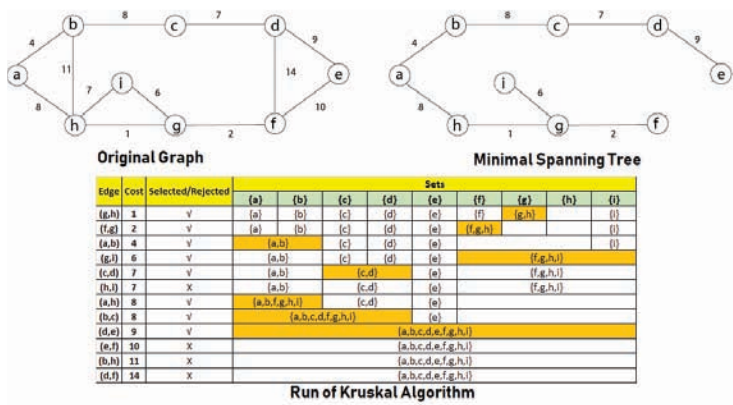
A disjoint set data structure maintains a collection $S = \{S_1, S_2, \dots, S_k\}$ of disjoint *dynamic* sets. Each set is identified by a representative, which usually is a member in the set.

- **Operations on Sets:** Let x be an object. We wish to support the following operations.
- **MakeSet(x)** creates a new set whose only member is pointed by x ; Note that x is not in the other sets. x is set as the leader of the set.
- **Union(x, y)** unites two dynamic sets containing objects x and y , say S_x and S_y , into a new set that $S_x \cup S_y$, assuming that $S_x \cap S_y = \emptyset$
- **FindSet(x)** returns a pointer to the representative/Leader of the set containing x .

ALGORITHM MST- Kruskal ($G, W[[]]$)

```
BEGIN:
Sort all Edges In E According To their Weights
Set of Edges of MST E' ← ∅
For all U ∈ V [G] DO
  MakeSet ( U )
For Each Edge(U , V) ∈ E [G] DO
  IF FindSet (U) ≠ FindSet (V)
    Union(U, V)
  Select(U, V) In E'
RETURN E'
END ;
```

Time Complexity: $O(|E|+|V|\text{Log } |V|)$
Space Complexity: $\Theta(|V|)$



17.8. TOPOLOGICAL SORT

Assume that we need to schedule a series of tasks, such as classes or construction jobs, where we cannot start one task until after its prerequisites are completed. We wish to organize the tasks into a linear order that allows us to complete them one at a time without violating any prerequisites. We can model the problem using a DAG. The graph is directed because one task is a prerequisite of another -- the vertices have a directed relationship. It is acyclic because a cycle would indicate a conflicting series of prerequisites that could not be completed without violating at least one prerequisite. The process of laying out the vertices of a DAG in a linear

order to meet the prerequisite rules is called a ***topological sort***.

There are two methods by which the topological sort sequence can be found:

- Using queue and indegree of nodes; and
- Using DFS.

Here we are going to use the Indegree of the nodes as the base for computing Topological sort sequence. 0 Indegree of a node shows that the activity represented by this node is independent and can be started immediately. Such nodes are inserted in the queue. We keep on exploring vertices from the first deleted node from the queue and decrease the connected node's indegree by 1. In this process, if 0 indegree node is found, it is added to queue and process repeats until the queue is empty.

ALGORITHM Topological Sort (G , Indeg [])

BEGIN:

Queue Q

Initialize (Q)

FOR All $U \in V [G]$

IF $\text{Indeg} [U] = 0$

Enqueue (Q, U)

WHILE ! Empty (Q) DO

$U \leftarrow \text{Dequeue} (Q)$

WRITE (U)

FOR All $V \in \text{Adj} [U]$ DO

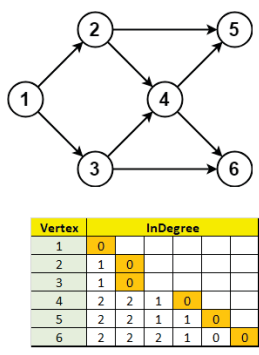
IF $\text{Indeg} [V] = 0$

Enqueue (Q, U)

END;

Time Complexity: $\Theta(|V|+|E|)$

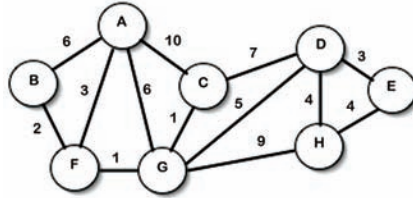
Space Complexity: $\Theta(|V|)$



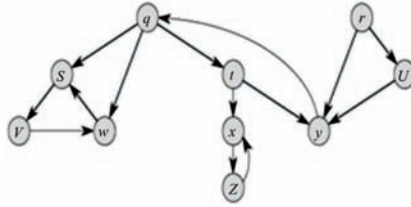
Queue	1					
Queue		2	3			
		TS 1				
Queue			3			
		TS 12				
Queue				4		
		TS 123				
Queue					5	6
		TS 1234				
Queue						6
		TS 12345				
Queue						
		TS 123456				

EXERCISES

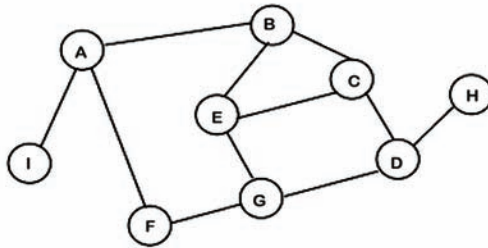
1. Find the minimal spanning tree of the graph given below using prim's method.



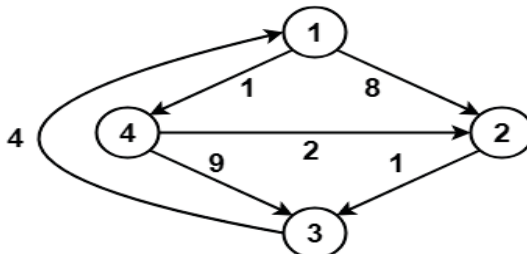
2. Traverse the given graph using DFS method. Draw the DFS forest.



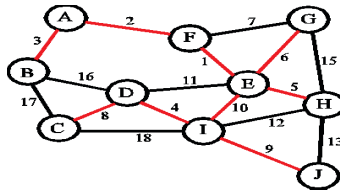
3. Write the algorithm for finding BFS in the graph. Find BFS sequence for the graph given below.



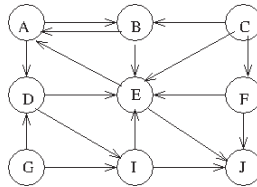
4. Find shortest path between every pair of vertex in the graph given below using Flloyd Warshall Algorithm.



5. Find the minimal spanning tree of the graph given below using prim's method.



6. Traverse the given graph using DFS method. Draw the DFS forest.



7. Run time of the Kruskal algorithm is
- $O(E \log E)$
 - $O(V \log E)$
 - $O(E \log V)$
 - $O(V+E)$
8. Data structure used by Prim's method is
- Linked list
 - Stack
 - Priority Queue
 - None
9. No of entries in the adjacency matrix of an undirected graph is
- $|E|$
 - $2|E|$
 - $|V| + |E|$
 - none of these
10. No of edges in a n -vertex complete graph is given by
- n^2
 - $n(n-1)/2$
 - n
 - $n-1$
11. Pick the odd one out
- Dijkstra's algo

- b. Prim's Algo
 - c. Huffman code
 - d. Inorder traversal of tree
- 12. Threaded representation without the header node has.....Null pointers**
- a. 2
 - b. 4
 - c. 0
 - d. none of these
- 13. Implicit binary tree representation suits**
- a. strictly binary tree
 - b. 2- Tree
 - c. extended tree
 - d. almost complete binary tree
- 14. A node with degree 0 in a graph is called**
- a. Vacant node
 - b. Eliminated node
 - c. Isolated node
 - d. None of these
- 15. A graph has edges $e_1=(u, u)$, $e_2=(u, v)$, $e_3=(u, v)$. This graph can be named as**
- a. Multigraph
 - b. Complete graph
 - c. Unconnected graph
 - d. None of these
- 16. In order traversal on a binary search tree yields keys in .**
- a. Decreasing order
 - b. Random order
 - c. Increasing Order
 - d. None of these
- 17. Tree based sorting technique has the complexity**
- a. $O(n^2)$
 - b. $O(n)$
 - c. $O(n \log n)$
 - d. none of these

18. A connected graph $G=(V, E)$ without any cycle contains

- $|V|$ edges
- $|V| - 1$ edges
- $|V| + 1$ edges
- $2 * |V|$ edges

Consider a B-tree of order m with n keys is

19. Search requires

- $O(\log_m n)$ time
- $O(\log n m)$ time
- $O(\log_2 m)$ time
- $O(\log_2 n)$ time

20. Minimum allowed keys in a node is

- $m - 1$
- $(m - 1)/2$
- 1
- $2m - 1$

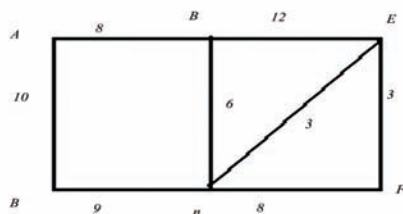
21. Maximum allowed keys in a node is

- $m - 1$
- $(m - 1)/2$
- 1
- $2m - 1$

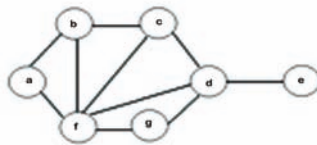
22. A graph $G=(V, E)$ has a spanning tree $T=(V, E)$ with $|E|$

- $|V|$
- $|V| - 1$
- $|V| + 1$
- $2|V| - 1$

23. Consider the graph given below. The shortest distance between A to F is.....

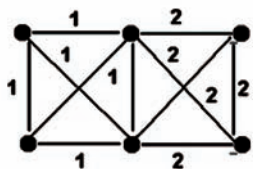


- a. 20
 - b. 15
 - c. 16
 - d. None of these
24. **Optimal merge pattern is an approach for building**
- a. Shannon Fano tree
 - b. Huffman Tree
 - c. AVL Tree
 - d. Heap
25. **Which of the following is not an application of Minimal Spanning tree**
- a. Communication Network
 - b. Electricity Network
 - c. Transportation Network
 - d. Decision Network
26. **Which among the following is not an application of graph**
- a. PCB
 - b. Web
 - c. DFD in Database
 - d. ER – Diagram in database
27. **Given a graph. Which of the following path is a simple path?**
- a. a f d g f c
 - b. a f g d c b
 - c. a b c f d c
 - d. a f d c d c



28. **The solution to the shortest path problem by Warshall's Algorithm for a Graph with n Vertices requires**
- a. $O(n^2)$
 - b. $O(n^2 \log_2 n)$
 - c. $O(n^2 \log_2 2n)$
 - d. $O(n^3)$

29. Finding BFS on a graph $G = (V, E)$ requires
Time ($|V|$ represents no of vertices and $|E|$ represents no of edges)
- a. $O(|V| + |E|)$
 - b. $O(|V| |E|)$
 - c. $O(|V| \log |E|)$
 - d. $O(|E| \log |V|)$
30. What is the cost of minimal spanning tree shown below?



- a. 10
 - b. 9
 - c. 8
 - d. 7
31. Using Kruskal's Algorithm for minimal spanning tree, find the edges of MST

	A	B	C	D	E
A		10	9	8	4
B			6	3	5
C				8	12
D					15
E					

- a. $[B, D], [A, E], [B, E], [B, C]$
 - b. $[D, C], [C, E], [B, A], [A, C]$
 - c. $[B, D], [A, E], [B, A], [B, C]$
 - d. $[B, D], [A, E], [B, E], [B, A]$
32. What is the cost of the spanning tree?
- a. 46
 - b. 18
 - c. 37
 - d. 45
33. Applying Warshall's Algorithm for shortest path in the graph with weight matrix given below, find the minimum cost between every

pair of vertices taking node A as mediator

	A	B	C	D
A	0	20	25	60
B	40	0	30	0
C	35	0	0	100
D	0	0	0	0

a.

	A	B	C	D
A	0	20	25	60
B	40	0	30	100
C	35	55	0	95
D	0	∞	∞	0

b.

	A	B	C	D
A	0	20	25	50
B	40	0	30	100
C	35	55	0	95
D	0	∞	∞	0

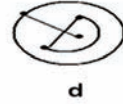
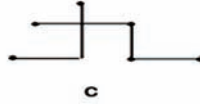
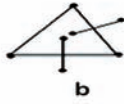
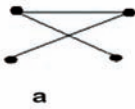
c.

	A	B	C	D
A	0	20	25	60
B	40	0	30	90
C	35	55	0	95
D	0	∞	∞	0

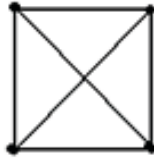
d.

	A	B	C	D
A	0	20	25	60
B	40	0	30	100
C	35	55	0	90
D	0	∞	∞	0

34. Consider the graphs given below, which of these are connected



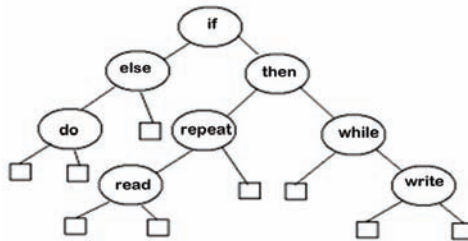
- i. a
 - j. b
 - k. c
 - l. d
35. How many spanning tree are possible of the graph given below



- a. 8
 - b. 16
 - c. 9
 - d. 12
36. Given a graph $G=(V, E)$ with $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ & $E = \{[a, b], [d, e], [j, k], [g, k], [e, f], [b, c], [i, k]\}$. Find the connected components of the graph.
- a. $\{a, b, c\}, \{d, e, f\}, \{g, i, j, k\}$
 - b. $\{a, b, c\}, \{d, e, f\}, \{g, i, j\}$
 - c. $\{a, b, c\}, \{d, e, f\}, \{g, i\}, \{j, k\}$
 - d. None of these
37. Which among the following defines a connected graph
- a. Path exists between every pair of vertices
 - b. Path exists between some pair of vertices
 - c. Path exists between single pair of vertices
 - d. None of the above
38. A transitive Closure of a graph is a matrix with respect to graph that contains
- a. Entries that tells the edges between every pair of vertices
 - b. All the path of length 1 between every pair of vertices
 - c. All the paths of different length

d. None of the above

39. External & Internal path length of the given tree is



a. 39, 22

b. 14, 30

c. 30, 14

d. None of these

18

Sparse Matrices

CONTENTS

18.1. Sparse Matrix Basics	434
18.2. Types Of Sparse Matrix	434
18.3. Addition Of Two Sparse Matrices	438
Exercises	439

18.1. SPARSE MATRIX BASICS

A sparse matrix or sparse array is a matrix in which most of the elements are zero. The number of zero-valued elements divided by the total number of elements (e.g., $m \times n$ for an $m \times n$ matrix) is called the sparsity of the matrix. Using those definitions, a matrix will be sparse when its sparsity is greater than 0.5.

When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix. Operations using standard dense-matrix structures and algorithms are slow and inefficient when applied to large sparse matrices as processing and memory are wasted on the zeroes. Sparse data is by nature more easily compressed and thus requires significantly less storage. Some very large sparse matrices are infeasible to manipulate using standard dense-matrix algorithms.

18.1.1. Storing a Sparse Matrix

A matrix is typically stored as a two-dimensional array. Each entry in the array represents an element $a_{i,j}$ of the matrix and is accessed by the two indices i and j . Conventionally, i is the row index, numbered from top to bottom, and j is the column index, numbered from left to right. For an $m \times n$ matrix, the amount of memory required to store the matrix in this format is proportional to $m \times n$ (disregarding the fact that the dimensions of the matrix also need to be stored).

In the case of a sparse matrix, substantial memory requirement reductions can be realized by storing only the non-zero entries. Depending on the number and distribution of the non-zero entries, different data structures can be used and yield huge savings in memory when compared to the basic approach.

18.2. TYPES OF SPARSE MATRIX

18.2.1. With Pattern

1	0	0	0	0		1	0	0	0	0		1	10	0	0	0
0	2	0	0	0		2	3	0	0	0		6	2	11	0	0
0	0	3	0	0		4	5	6	0	0		0	7	3	12	0
0	0	0	4	0		7	8	9	10	0		0	0	8	4	13
0	0	0	0	5		11	12	13	14	15		0	0	0	9	5

Diagonal Sparse Matrix

Tirangular Sparse Matrix

Tridiagonal sparse Matrix

$$= i*(i-1)/2$$

Before $\langle i, j \rangle$ element in i th column, element in $j-1$ columns would already have been placed.

$$\text{Total Elements before } \langle i, j \rangle = i*(i-1)/2 + (j-1)$$

$$\text{For storage of } \langle i, j \rangle \text{ location in one dimensional array} = i*(i-1)/2 + (j-1) + 1$$

- **Tridiagonal Sparse Matrix:** Consider a diagonal sparse matrix given below.

1	1	0	0	0	0	0
1	2	2	0	0	0	0
0	2	3	3	0	0	0
0	0	3	4	4	0	0
0	0	0	4	5	5	0
0	0	0	0	5	...	y
0	0	0	0	0	x	N

To save memory, we can use one-dimensional array. Array index for $\langle i, j \rangle$ element of the Matrix can be found with the following method

There are three elements in each row except the first and last row in which there are two elements

Before storing $\langle i, j \rangle$ element in the One-dimensional array, elements upto $i-1$ rows would already have been placed.

$$\text{No of elements upto } i-1 \text{ rows} = 2 + (3+3+ \dots + i-2 \text{ times})$$

$$= 2 + 3*(i-2)$$

In each row there are $j-i+1$ element before $\langle i, j \rangle$ entry

For storage of $\langle i, j \rangle$ element, location in one dimensional array

$$= 2 + 3*(i-2) + j - i + 1 + 1$$

$$= 2 + 3i - 6 + j - i + 2$$

$$= 2i + j - 2$$

- **Storage of Sparse Matrices with No Pattern:** Sparse matrices with no pattern can be stored with two methods:
 - Array of structure; and
 - Linked list.
- **Array of Structure Representation of Sparse Matrix:** This representation is also called *Vector Representation*. For the storage of elements, information about their row & column number is a must as it will help us in carrying out computations.

Struct sparse

```
{
int element;
int row;
int column;
};
```

Struct sparse S[10];

		Column										
		1	2	3	4	5			Row	Column	Element	
Row	1	0	0	7	0	9		S[0]	1	3	7	
	2	3	0	0	4	0		S[1]	1	5	9	
	3	0	8	3	0	0		S[2]	2	1	3	
	4	0	0	9	0	4		S[3]	2	4	4	
	5	1	0	0	0	12		S[4]	3	2	8	
								S[5]	3	3	3	
								S[6]	4	3	9	
								S[7]	4	5	4	
								S[8]	5	1	1	
								S[9]	5	5	12	

- **Linked List Representation:** For Linked list representation, node used to store the sparse matrix element is supposed to have 4 fields

row	column	element	next
-----	--------	---------	------

		Column				
		1	2	3	4	5
Row	1	0	0	7	0	9
	2	3	0	0	4	0
	3	0	8	3	0	0
	4	0	0	9	0	4
	5	1	0	0	0	7

S	1	3	7			1	5	9			2	1	3			2	4	4			3	2	8			3	3	3			4	3	9			4	5	4			5	1	1			5	5	7	\
---	---	---	---	--	--	---	---	---	--	--	---	---	---	--	--	---	---	---	--	--	---	---	---	--	--	---	---	---	--	--	---	---	---	--	--	---	---	---	--	--	---	---	---	--	--	---	---	---	---

18.3. ADDITION OF TWO SPARSE MATRICES

Suppose two sparse matrices are given as the linked lists. For addition to be performed, row, and column numbers of the matrix should be same for the current node. The logic for the addition is given in the Algorithm shown below

ALGORITHM SparseMatrixAddition(Sparse1, Sparse2)

BEGIN:

p=Sparse1

q=Sparse2

Sparse3=NULL

WHILE p!=NULL AND q!=NULL DO

IF p→row == q→row THEN

IF p→column == q→column THEN

InsBeg(Sparse3, p→row, p→column,
p→element+q→element)

ELSE

IF p→column < q→column THEN

InsBeg(Sparse3, p→row, p→column,
p→element)

ELSE

InsBeg(Sparse3, q→row, q→column,
q→element)

ELSE

IF p→row < q→row THEN

InsBeg(Sparse3, p→row, p→column,
p→element)

ELSE

InsBeg(Sparse3, q→row, q→column,
q→element)

WHILE p!=NULL DO

InsBeg(Sparse3, p→row, p→column, p→element)

WHILE q!=NULL DO

InsBeg(Sparse3, q→row, q→column, q→element)

RETURN Sparse3

END;

EXERCISES

1. **Write a Program in C for finding Transpose of a Sparse Matrix.**
2. **Write a Program in C for multiplying two Sparse Matrices**
3. **What are the various ways a sparse matrix can be represented to save the storage space?**
4. **Write a program to find if the given matrix is sparse?**
5. **We store a triangular sparse array of size $n \times n$ in a two dimensional array storing only the nonzero elements. The base address of such an array is 1000, what will be the address of element $A[100, 50]$? (consider every element occupies 1 word of m/y).**
6. **Pick the odd one out**
 - a. Adjacency matrix of the tree graph
 - b. Identity matrix
 - c. Tridiagonal matrix
 - d. Adjacency matrix of the complete graph

19

Hashing

CONTENTS

19.1. Direct Address Table	442
19.2. Hash Tables	442
19.3. Collision Resolution In Hash Table.....	444
Exercises.....	448

In Compilers, during Lexical analysis phase various tokens are identified and saved in the symbol table. During the execution of the program it will be required to see the type of the variable whenever it will appear in the statements for computations. If the program is long, reference to variables will be frequent. We require a technique which stores the tokens in the symbol table such that the retrieval is achieved in quick time (Constant time preferably).

Consider a situation where names are associated with the vertices of the graph. Computation in the graph are done considering vertex number. Hence a quick conversion of name to vertex number is required.

Hashing is a method that will provide us with the searching in constant time. Data structures used for symbol table is the Hash Table.

19.1. DIRECT ADDRESS TABLE

Suppose we want to store the marks of 60 students of a class in a particular subject. Let us consider an array of size 60.

	[1]	[2]	[3]	[4]	...														[60]
Marks																			

In the array,
Marks [1] represents the marks of student with roll no 1,
Marks [2] represents the marks of student with roll no 2,
Marks [3] represents the marks of student with roll no 3,
...
Accessing marks of one students in such table can be performed in constant time. Marks of roll No i student is Marks[i]. Such tables in which the key value is treated as the index, is called the Direct Address Table.

19.2. HASH TABLES

Hash tables are like the direct address table. Usually, the Hash table size is huge and the key values may not tell the direct index in the array hence a function is applied to convert the key to index in the Hash Table.

H:Key→L
H is the hash function, L is the Location of the key in the hash table
There are various hash functions possible:

- Division hash function;
- Mid-square function;
- Folding hash function.

Consider a hash table of size 100 with indexes ranging from 0–99. We required the Hash function to generate the address from 00–99. For this purpose the modulus of key with 100 can be taken. This will generate the addresses in between 00–99.

Since 100 has many factors, there is a possibility of getting the same remainder upon division with 100. e.g., 12345, 23445, 32345, all will result in the same location i.e., 45. To avoid getting the same remainder, We can divide the key with the prime number near to 100 (97 e.g.,). Division with the prime number will decrease the possibility of the repeated remainder. There will be some locations (97, 98, 99 in this example) in the Hash Table which will remain unutilized.

key	Location in Hash Table	
	key%100	key%97
12345	45	26
78399	99	23
54277	77	54
80290	90	71
53535	35	88

19.2.1. Mid-Square Method

Consider a hash table of size 100 with indexes ranging from 0–99. We required the Hash function to generate the address from 00–99. Key is squared in this method and middle two digits are selected as the hash address.

key	Key ²	Hash Address
12345	152399025	39
78399	6146403201	40
54277	2945992729	99
80290	6446484100	48
53535	2865996225	99

19.2.2. Folding Method

Consider a hash table of size 100 with indexes ranging from 0–99. We required the Hash function to generate the address from 00–99. Key is divided into groups of two-digit numbers starting from least significant digit (LSD). All groups are added. If any carry is generated, it is discarded. Remaining is the address of the key in the Hash Table

key	Groups	Group Sum	Hash Address
1803250049	18+03+25+00+49	95	95
1803210148	18+03+21+01+48	91	91
1803240117	18+03+24+01+17	63	63
1803231901	18+03+23+09+01	54	54
1803200083	18+03+20+00+83	124	24

19.3. COLLISION RESOLUTION IN HASH TABLE

If the location found for the two keys are same, this is called collision.

$H:Key_1 \rightarrow L$

$H:Key_2 \rightarrow L$

Collision is not a desirable situation in Hashing. Avoiding collisions completely is hard, even with a good hash function. Some method should be used to resolve this.

There are two methods of collision resolution in Hashing.

- Chaining
- Open Addressing

19.3.1. Chaining

- Chaining method takes the array of linked list in contrast to linear array for Hash Table. The keys with same hash address goes to same linked list. This way a key never goes in the wrong bucket.
- If all n keys hash to the same slot, worst case occurs. Worst-case time to search is $O(n)$, plus time to compute the hash function

Load Factor of a Hash Table

- Load factor of a hash table T :

$$\alpha = n/m$$

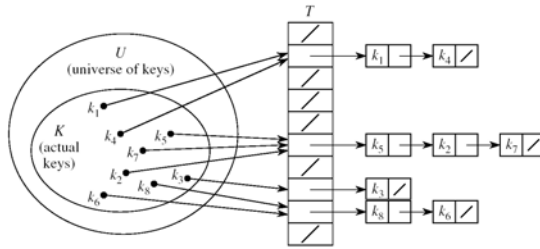
where;

n = number of elements stored in the table

m = number of slots in the table = number of linked lists

α encodes the average number of elements stored in a chain

α can be less than, equal to or greater than 1



19.3.2. Open Addressing

If we have enough contiguous memory to store all the keys ($m > N$) $\Rightarrow$ store the keys in the table itself rather than using link in the same cell.

Basic idea:

- **Insertion:** If a slot is full, try another one, until you find an empty one.
- **Search:** Follow the same sequence of probes.

Search time depends on the length of the probe sequence.

Open addressing methods:

- Linear probing
- Quadratic probing
- Double hashing/ Rehashing
- Linear Probing:

Idea: when there is a collision, check the next available position in the table (i.e., probing)

$$h(k, i) = (h_1(k) + i) \bmod m$$

$i=0, 1, 2, \dots$

- First slot probed: $h_1(k)$
- Second slot probed: $(h_1(k) + 1) \bmod m$
- Third slot probed: $(h_1(k) + 2) \bmod m$, and so on

Modulus is applied because the Hash table is considered to be circular in nature.

- The possible cases at given hash location are:
 1. The Location in the Hash table is occupied by an element with the key equal to the given key
 2. The Location in the Hash table is empty
 3. The Location in the Hash table is occupied by the different key
- Search the next higher index in the Hash Table until the empty position is discovered

1. Quadratic Probing

Idea: when there is a collision, check the next available position in the table

using the quadratic formula:

$$h(k, i) = (h_1(k) + a*i + b*i^2) \bmod m$$

$i=0, 1, 2, \dots$

if $a=1/2, b=1/2$

- First slot probed: $h_1(k)$
- Second slot probed: $(h_1(k) + \frac{1}{2} + \frac{1}{2}) \bmod m$
- Third slot probed: $(h_1(k) + \frac{1}{2}*2 + \frac{1}{2}*2^2) \bmod m$ and so on
- Fourth slot probed: $(h_1(k) + \frac{1}{2}*3 + \frac{1}{2}*3^2) \bmod m$ and so on

$h_1(k), h_1(k)+1, h_1(k)+3, h_1(k)+6, \dots$

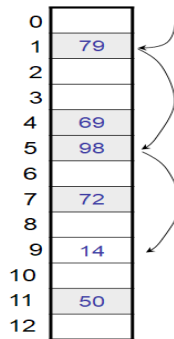
2. Double Hashing

Use one hash function to determine the first slot and use a second hash function to determine the increment for the probe sequence

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m, i=0, 1, \dots$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on...

Example



$$h_1(k) = k \bmod 13$$

$$h_2(k) = 1 + (k \bmod 11)$$

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod 13$$

- Insert key 14:

$$h_1(14, 0) = 14 \bmod 13 = 1$$

$$h(14, 1) = (h_1(14) + h_2(14)) \bmod 13$$

$$= (1 + 4) \bmod 13 = 5$$

$$H(14, 2) = (h_1(14) + 2 h_2(14)) \bmod 13$$

$$= (1 + 8) \bmod 13 = 9$$

3. Average Probes Required for Successful Search

Consider a hash table of size 13

0	1	2	3	4	5	6	7	8	9	10	11	12

Consider insertion of following keys using linear probing

0	1	2	3	4	5	6	7	8	9	10	11	12
H	C	J		B	A	G	D		F		E	I

Total No of probes required for successful search = $1+1+1+1+1+1+3+1+2+4$

Average No of probes required for successful search = $16/10 = 1.6$

4. Average Probes Required for Unsuccessful Search

For the above Hash Table, if the key being searched is not there in the table,

Total No of probes requires, looking at each table location, is equal to $4+3+2+1+5+4+3+2+1+2+1+6+5 = 39$

Average number of probes required for unsuccessful search = $39/13 = 3$

Both, Average probes for successful search and unsuccessful searches are constant irrespective of the table size.

Hashing method produces better result in terms of average case behavior of the search performance in comparison to Linear and Binary search. Hash Table can be used at the places where fast look up table is required.

EXERCISES

1. Given the values {2341, 4234, 2839, 430, 22, 397, 3920}. Consider a hash table of size 7, and hash function $h(x) = x \bmod 7$. Show the resulting tables after inserting the values in the given order with each of these collision strategies.
 - a) Linear probing
 - b) Quadratic probing
 - c) Double hashing with $(2x-1) \bmod 7$
2. Suppose the table T has 11 memory locations T[1], T[2], ..., T[11]. A file consists of 8 records A, B, C, D, E, X, Y, Z W. Following are the hash addresses:
 Record: A, B, C, D, E, X, Y, Z
 H(K): 4, 8, 2, 11, 4, 11, 5, 1
 If records are stored in the order Z, Y, X, E, D, C, B, A, find out the memory map of storage of records in the file.
 If keys are searched after storage, what will be average number of probes for successful and Unsuccessful searches?
3. Collision in the Hash Table occurs if
 - Two entries with different data have the exact same key.
 - Two entries with different keys have the same exact hash value.
 - Two entries are identical except for their keys.
 - Two entries with the exact same key have different hash values.
4. A Hash table with chaining has been declared with 512 sized array of linked list. The maximum keys that can be placed in this table is
 - 512
 - 256
 - 511
 - No upper cap on number of elements
5. A hash table of length 10 uses open addressing with hash function $h(k)=k \bmod 10$, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34

5	52
6	46
7	33
8	
9	
10	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

- 46, 34, 42, 23, 52, 33
 - 46, 42, 34, 52, 23, 33
 - 42, 46, 33, 23, 34, 52
 - 34, 42, 23, 52, 33, 46
6. If the following keys are to be accommodated in the hash table with Hash Function ' $x \bmod 10$, ' Which among the following statement is true?
- i) 9679, 1989, 4199 hash to the same value
 - ii) 1471, 6171 has to the same value
 - iii) All elements hash to the same value
 - iv) Each element hashes to a different value
- i only
 - ii only
 - i and ii only
 - iii or iv
7. Given a Hash table of size 7 (indexed from 0 to 6). A hash function is defined as $(3x + 4) \bmod 7$ for storage of keys in the Hash Table. Which of the following represents the Hash table structure after storage of the numbers 1, 3, 8, 10 in the sequence using closed hashing? Note that ' $_$ ' denotes an empty location in the table.

*	1					3
*	1	10	8			3
*	8					10
*	1	8	10			3

8. Design a C Program for computation of addresses for various Hash functions.
9. Design a C Program for collision resolution in Hash Table using

Linear Probing Method.

10. **Design a C Program for collision resolution in Hash Table using Quadratic Probing Method.**
11. **Design a C Program for collision resolution in Hash Table using double hashing Method.**
12. **Design a C Program for Implementation of Chaining Method in Hashing.**
13. **If searching is performed on the ordered table of size n where the keys are uniformly distributed between $\text{Key}(1) - \text{Key}(n)$, which of the following techniques will require least effort**
 - a. Sequential search
 - b. Index sequential search
 - c. Binary search
 - d. Interpolation search
14. **Pick the odd one out**
 - a. Open addressing
 - b. Separate chaining
 - c. Linear Hashing
 - d. Rehashing

INDEX

A

Activation records 342, 343, 350,
351, 352, 353, 354, 356, 375
Adjacency matrix 406
Adjust Algorithm 99
Algorithm builds Huffman 358
Algorithmic process 46
Appropriate position 58, 68
Arithmetic expression 158
Array implementation 217
Ascending priority queue 104
Asymptotic notation 29

B

Binary Search Tree (BST) 372
Boolean value function 208, 212
Breadth-first search (BFS) 206, 407

C

Circular Queue 209, 212, 213, 222
Complete binary tree 339

Complexity 22, 24, 25, 26, 27, 28,
29, 30, 100, 104, 105, 106
Computer resources 22
Computer science application 124
Conquer algorithm 67

D

Data compression 338
Data elements 226
Data elements construct 8
Data structure 2, 4, 6, 8, 9, 124, 226,
333
Data structures 434
Dense-matrix algorithm 434
Dense-matrix structures 434
Depth-first search (DFS) 407
Dimensional array 434, 436, 439
Directed Graph 404

E

Efficiency measure 22
Element 124, 129

Element automatically 67
 Execute statements 154, 156, 160
 Extreme left element 372, 373

F

Factorial function 188
 First Come First Serve (FCFS) 2
 First-in-first-out (FIFO) 2, 206
 Function definition 23, 24

H

Hard Disk Sector 396, 397
 Hash Function 449
 Hash Table 443, 444, 445, 447, 448, 449, 450
 Heap property 103
 Hexadecimal Conversion 145
 Hierarchical relationship 8
 Highest Priority element 217
 Huffman Coding 358

I

Implementation 340, 341
 Infix Expression 170, 171, 175, 176, 177, 178
 Inserted element 102
 Internal node 397

L

Last-in-first-out (LIFO) 4
 Least significant digit (LSD) 443, 94
 Lexical analysis phase 442
 Linear collection 4, 226
 Linear placement 226
 Linear queue 206, 222, 223, 224
 Lossless text compression Technique 358
 Lower priority elements 217

Low priority process 217

M

Matrix Element 122
 Matrix product 122
 Matrix transposition 114, 115
 Median key 395, 397
 Merge Operation 80

N

Nonnegative integer 91

O

One-dimensional array 435, 436

P

Polish notation expression 163
 Pop operation 140
 Postfix expression 169
 Postfix Expression 163, 170, 171, 172, 176, 177
 Power function 190
 Primitive operation 140, 149, 150
 Priority Queue 358, 359, 104, 217, 218, 220

R

Recursion 187, 188, 189, 191, 194, 199, 202
 Recursive function 238
 Right element 74

S

Search performance 447
 Similar arrangement 196
 Space complexities 29
 Space Complexity 22, 25, 26, 27
 Sparse matrices 434, 438
 Sparse matrix or sparse array 434

Square Matrix 406

Standard Algorithms 406

Structure Declaration 149, 150

Substantial memory requirement

434

Symbol decision 168

T

Telephone directory 46

A Practical Approach to Data Structure and Algorithm with Programming in C

"An Algorithm must be seen to be believed"

-Donald Knuth

The purpose of this book is to culminate a deep understanding of Data Structures and Algorithms among the students. The book provides a thorough and comprehensive coverage of the fundamentals of data structures and the principles of algorithm analysis. The book introduces the concept of basic data structures, numerous data structure techniques used in the current paradigm, and the basics of algorithms. A structured approach is followed to explain the process of problem-solving. A theoretical description of the problem is followed by the underlying technique.

The book then proceeds to explain in detail each of the data structures and elaborates the methods associated with it. A detailed explanation is provided with each method, the algorithm, and the C code. These are then ably supported by an example followed by an algorithm, and finally the corresponding program in C language.

In conclusion, the aim of this book is to provide a deep understanding of all data structures and algorithms, and all the methods associated with the implementation and operation of all the Data Structures. The book tries to explain concepts in-depth, starting from very basic.

This book is aimed at serving engineering graduate students of computer science and postgraduate level courses of computer applications. This book also works as a catalyst to the students approaching for competitive examinations such as GATE, UGC, NET, PSUs, etc.



Mr. Akhilesh Kumar Srivastava is a Computer Science Graduate from K.N.I.T. Sultanpur and Post Graduate from Dr APJ Abdul Kalam Technical University Lucknow. Author is GATE and UGC Net Qualified. He has written 3 Books and several research Papers in International/ National Journals, presented papers in International Conferences. Author runs his own You Tube Educational Channel with substantial viewership across the globe. Author has also received several awards from Academic institutes and Industries like Infosys Ltd.